

**PRIORITY  
DOCUMENT**

SUBMITTED OR TRANSMITTED IN  
COMPLIANCE WITH RULE 17.1(a) OR (b)



REC'D 29 OCT 2003

WIPO

PCT

**Prioritätsbescheinigung über die Einreichung  
einer Patentanmeldung**

**Aktenzeichen:** 102 46 369.7

**Anmeldetag:** 30. September 2002

**Anmelder/Inhaber:** Dr. Thomas Schöbel-Thœuer, Stuttgart/DE

**Bezeichnung:** Verfahren zur Regelung des Datenzugriffs auf Daten  
wenigstens einer Datenspeichereinrichtung

**IPC:** G 06 F 15/167

**Die angehefteten Stücke sind eine richtige und genaue Wiedergabe der ursprünglichen Unterlagen dieser Patentanmeldung.**

München, den 8. Oktober 2003  
Deutsches Patent- und Markenamt  
Der Präsident  
Im Auftrag

Schöbel

BEST AVAILABLE COPY



# Inhaltsverzeichnis

<b>1</b>	<b>Ziele und Grundidee</b>	
1.1	Einführung	4
		4
<b>2</b>	<b>Entwurfs-Prinzipien</b>	
2.1	Generizität: Wahl der angemessenen Abstraktionsebene	7
2.1.1	Parametrische Generizität	7
2.1.2	Erweiterungs-Generizität	7
2.1.3	Kompositorische Generizität	8
2.1.4	Universelle Generizität	8
2.1.5	Fragen der Anwendung von Generizität	9
	Trennung von Mechanismus / Strategie / Repräsentation	9
	Das Verantwortungs-Prinzip	11
2.3.1	Kontrollflüsse	11
2.3.2	Schnittstellen-Mechanismen	11
2.3.3	Abgrenzung von Verantwortung durch Kapselung	12
2.4	Zerlegungs- und Rekombinationsstrategien	13
2.4.1	Zerlegung der zu lösenden Aufgaben in möglichst wenige universelle Elementarteile	13
2.4.2	Orthogonalität und Rekombinierbarkeit von Elementaroperationen	13
2.5	Das Problem der „Eierlegenden Wollmilchsau“	13
2.6	Automatismen	14
2.7	Zugriffsrechte und Schutzmechanismen	14
2.7.1	Grundlegende Betrachtung	15
2.7.2	Spezifikation von Schutzrechten	15
2.7.3	Prüfung von Schutzrechten	16
2.7.4	Sicherstellung von Schutzrechten	16
2.8	Performanz-Fragen	16
2.8.1	Performanz von Elementaroperationen	16
2.8.2	Zero-Copy-Architektur	17
2.8.3	Das Background-IO-Konzept	17
		17
<b>3</b>	<b>Nester</b>	
3.1	Arten von Nester	19
3.2	Zugriffs-Modelle	19
3.3	Elementaroperationen auf statischen und dynamischen Nester	20
3.3.1	transfer	22
3.3.2	wait	23
3.3.3	get	24
3.3.4	put	24
3.3.5	lock und unlock	25
3.3.6	Freiwillige Selbstkontrolle des Verhaltens	25
3.3.7	get_maxlen und set_maxlen	27
3.3.8	Stream-IO, Pipes, Sockets und Speicherverwaltung	27
3.4	Elementaroperationen auf dynamischen Nester	28
3.4.1	get_map	29
3.4.2	clear	29
3.4.3	delete	30
3.4.4	move	30
3.4.5	get_meta	30
		31

<b>4 Bausteine</b>	32
4.1 Beispiel-Baustein-Arten	32
4.1.1 device_*	33
4.1.2 buffer	33
4.1.3 map_*	34
4.1.4 selector	36
4.1.5 dir_*	36
4.1.6 union	38
4.1.7 mmu_*	38
4.1.8 adaptor_*	39
4.1.8.1 Adaption zwischen verschiedenen Zugriffs-Modellen	40
4.1.8.2 Adaption zwischen verschiedenen transfer_size-Werten	41
4.1.9 redirect	42
4.1.10 cow	42
4.1.11 transaction	43
4.1.11.1 Sequentielle Transaktionen	43
4.1.11.2 Parallele Transaktionen	43
4.1.12 remote	44
4.1.13 mirror	44
4.1.14 pipe	45
4.2 Instantiierung von Bausteinen	45
4.2.1 Der Mechanismus	45
4.2.2 Einige mögliche grundlegende Strategien	47
<b>5 Callbacks durch notify-Operationen</b>	49
5.1 Eigentums- und Besitzverhältnisse	49
5.2 Rückforderung von Eigentum	50
5.3 Synchronisation zwischen Kopien: spekulative Locks	52
<b>6 Generische Operationen</b>	55
6.1 Beispiel-Typsysteme	56
6.1.1 Minimal-Typsystem	56
6.1.2 Einfaches erweiterbares Typsystem	57
6.2 Zusammenhang mit Objektorientierung	58
6.3 Zusammenhang mit dem Funktionalen Paradigma	59
6.4 Zusammenhang mit Datenbank-Schemata	60
<b>7 Multiversion-Modelle</b>	62
7.1 Anforderungen	62
7.2 Allgemeines Modell	62
7.3 Zeitintervall-Modell	63
7.4 Container-Operationen: Locks	64
7.5 Kausale Abhängigkeiten	65
7.6 Aktualitätsgrade	65
7.7 Schnittstellen von Lock-Operationen	66
7.7.1 Suchintervalle	66
7.7.2 Lock-Operationen im Multiversion-Modell	66
7.8 Simulation einiger Transaktions-Strategien	68
7.8.1 Timestamp-Ordering-Protokolle	68
7.8.2 Striktes 2-Phasen-Locking	69
7.8.3 Weitere Arten von Serialisierbarkeit bzw. Konsistenzmodellen	69
7.8.4 Automatische Re-Evaluation	69
<b>A Simulation von Vererbung durch Generizität</b>	70
<b>B Beispiel-Probleme der Objektorientierung</b>	73
<b>C Ansätze zur Vermeidung von Wettrennen</b>	74
<b>Literaturverzeichnis</b>	75

# Kapitel 1

## Ziele und Grundidee

Die neue Betriebssystem-Architektur soll eine vollständige Verschmelzung von Datenbanken und Betriebssystemen ermöglichen, sowie den Funktionsumfang von verteilten Betriebssystemen, Netzwerk-Betriebssystemen und Echtzeit-Betriebssystemen, mit oder ohne MMU-Hardware, auf einer gemeinsamen Code-Basis ermöglichen.

Der Schlüssel dazu ist eine universelle Erweiterbarkeit und dynamische Konfiguration zur Laufzeit, wodurch sich die neue Architektur prinzipiell für jedes dieser Anwendungsbereiche eignet. Darüber hinaus sollen neue Anwendungsbereiche durch Konfigurationen erschlossen werden, die weder mit bisherigen Betriebssystemen noch mit Datenbanken auf einfache Weise realisierbar waren.

Erreicht wird dieses Ziel durch die Abstraktionen *Nest* und *Baustein*. Es wird gezeigt, dass sich ein performantes Betriebssystem im Wesentlichen allein aus diesen beiden Abstraktionen aufbauen lässt.

Gegenüber aktuellen Betriebssystem- und Kern-Architekturen wird die innere Komplexität in vielen Fällen reduziert, da eine relativ kleine Menge von primitiven Grundoperationen sehr weite Bereiche der wesentlichen Grundfunktionen eines Betriebssystems abdeckt; nicht nur Konzepte wie Inodes und deren Caches werden durch die Konzepte *Nest* und *Baustein* ersetzt, sondern sogar bisher als absolut notwendig erachtete Konzepte wie Dateisysteme als solche.

Die vorgestellte Architektur schreibt keine speziellen Implementierungs-Paradigmen vor; es sind sowohl Implementierungen durch Server-Prozesse („Communicating Sequential Processes“, z.B. in Mikrokern-Architekturen), als auch Implementierungen durch Rollenwechsel von Benutzer-Prozessen in Kern-Modi, als auch in Mischformen dieser Paradigmen möglich.

### 1.1 Einführung

Die Grundidee dieser Arbeit besteht darin, ein Betriebssystem auf allen Ebenen und Schichten im wesentlichen aus zwei Abstraktionen aufzubauen: *Nester* und *Bausteine*.

Präzisere Definitionen folgen später; zum Verständnis dieser Einführung reicht es, sich eine *Nest*-Instanz als einen (virtuellen) Adressraum vorzustellen, in dem sich Daten befinden können. Nicht alle Stellen dieses Adressraums müssen zugreifbar sein, es können beliebige Löcher darin vorhanden sein. Dies ähnelt einem Sparse File unter Unix, nur mit dem Unterschied, dass Zugriffe auf Löcher als Feh-

ler behandelt werden. In Erweiterung zu bisher verwendeten Adressraum-Abstraktionen gibt es auf Nestern eine Verschiebe-Operation *move*, mit der ein ganzer Speicherblock im virtuellen Adressraum auf virtuelle Weise verschoben werden kann. Der Speicherblock erscheint anschließend unter verschobenen virtuellen Adressen; die Implementierung erfolgt jedoch so, dass keine tatsächlichen Kopieroperationen im Speicher ablaufen, sondern nur der *virtuelle Eindruck* einer Verschiebung entsteht.

*Bausteine* kann man sich als „Transformatoren“ vorstellen, die ein oder mehrere Eingangs-Nester in ein oder mehrere andere Ausgangs-Nester transformieren. Bausteine lassen sich *instantiiieren*, d.h. man kann beliebig viele Inkarnationen des gleichen Baustein-Typs herstellen, deren Eingänge sich mit den Ausgängen anderer Baustein-Instanzen verbinden oder „verdrahten“ lassen. Als für Menschen leicht verständliche Darstellung benutze ich Zeichnungen, wie sie in der Elektro- und Digitaltechnik für Schaltbilder verwendet werden. Eine Baustein-Instanz wird als Kästchen mit linksseitigen Eingängen und rechtsseitigen Ausgängen gezeichnet. Die „Verdrahtungsregeln“ sind gleich wie bei Schaltbildern in der Elektrotechnik.

Als erstes Beispiel sehen wir uns in Abbildung 1.1 ein Szenario an, wie es bei konventionellen Betriebssystemen auftritt. Am Ende der Baustein-Hierarchie steht eine Instanz von *mmu\_i386*, die einen (nicht eingezeichneten) virtuellen Benutzer-Adressraum 1:1 auf ihr Eingangs-Nest abbildet, indem sie die MMU-Hardware (Memory Management Unit) benutzt. Der Benutzer-Adressraum enthält genau das, was sich im Eingangs-Nest dieses Bausteins „befindet“ bzw. was dort vom Vorgänger-Baustein (virtuell) zur Verfügung gestellt wird. In Beispiel ist dies ein virtuelles Prozessabbild, das von der *union*-Instanz generiert wird und als wesentliche Grundelemente jeweils ein Code-, ein Daten-, ein Stack-„Segment“ sowie eine virtuelle Abbildung einer *mmap*-Datei zu einem ausführbaren Prozessabbild zusammenstellt. Der Begriff „Segment“ wird jedoch in dieser Arbeit nicht verwendet, da er mit der Nest-Abstraktion zusammenfällt bzw. einen Spezialfall davon darstellt (üblicherweise enthält ein Segment lediglich keine Löcher).

Der *union*-Baustein wird aus mehreren Quellen gespeist. Drei davon sind *Nest*-Instanzen, die man konventionell als „Dateien“ oder „Files“ bezeichnen würde: eine Quelle ist eine *device\_ramdisk*-Instanz, die flüchtigen Hauptspeicher bereitstellt (in der Praxis ist es sinnvoll, hierfür eine gepufferte „temporäre Datei“ zu verwenden, damit

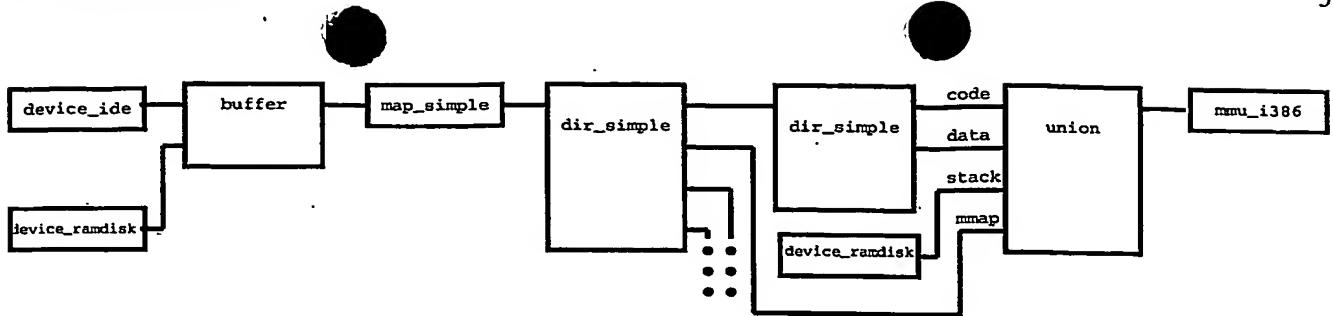


Abbildung 1.1: Erstes Beispiel

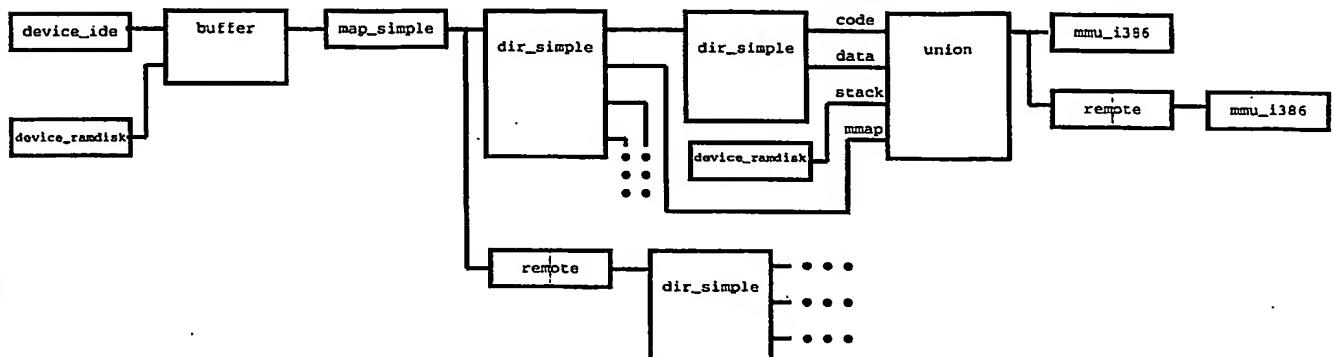


Abbildung 1.2: Zweites Beispiel

bei Speichermangel ein Auslagern auf Hintergrundspeicher möglich ist).

Die „Datei“-Nester stammen wiederum aus Quellen, die die Bezeichnung `dir_*` tragen. Die `dir_*`-Bausteine erfüllen ungefähr die gleichen Funktionen wie *Verzeichnisse* in konventionellen Dateisystemen. Im Unterschied zu konventionellen Dateisystemen implementieren sie keine *Verzeichnis-Bäume*, sondern nur flache Verzeichnisse. Durch Schachtelung von `dir_*`-Instanzen lassen sich sehr leicht konventionelle Verzeichnisbaum-Hierarchien nachbilden. Man kann sich eine `dir_*`-Instanz als eine Art „Container“ vorstellen, der den Platz für seine Ausgangs-Nester in seinem Eingangs-Nest allokiert und verwaltet. Dabei wird die Tatsache ausgenutzt, dass das Eingangs-Nest die relativ billige Verschiebeoperation `move` bereitstellt, mit deren Hilfe das *Platzmanagement* leicht lösbar wird. Verschiebe-Operationen können beispielsweise benutzt werden, wenn neue Ausgangs-Nester hinzukommen oder wenn sich die Größe oder der Platzbedarf eines Ausgangs-Nestes ändert.

Die billige virtuelle Verschiebeoperation muss irgendwo herkommen und implementiert werden; dies erledigt die `map_simple`-Instanz, die im Beispiel an der „Wurzel“ der Verzeichnis-Hierarchie steht. Dort werden diejenigen Probleme gelöst, die bei konventionellen Dateisystemen als *Fragmentierungs-Probleme* bekannt sind (Lokalitätsverhalten des Zugriffs).

Die vorgeschaltete `buffer`-Instanz sorgt für die Entkoppelung des zeitlichen Zugriffsverhaltens auf langsame Peripheriegeräte wie z.B. Festplatten (`device_ide`), und lässt sich von der Funktionalität her mit konventionellen

Buffer-Caches vergleichen.

Sämtliche Leitungen in der Zeichnung repräsentieren Nest-Instanzen, die jeweils immer die gleiche Schnittstelle besitzen. Die Bausteine sind daher in beinahe beliebiger Weise miteinander kombinierbar (die Frage ist nur, welche Kombinationen Sinn machen).

Das Beispiel in Abbildung 1.2 soll die Flexibilität dieses Systems andeutungsweise demonstrieren. Im Vergleich zu vorhin kommen zwei `remote`-Instanzen hinzu, die das Client-Server-Paradigma auf Nester übertragen. Eine `remote`-Instanz macht ein Nest so auf einem anderen Rechner verfügbar, als wäre es dort lokal vorhanden. Die eine `remote`-Instanz sitzt im Beispiel am Ende der Hierarchie und macht ein komplettes Prozessabbild auf einem anderen Rechner verfügbar. Dort befindet sich eine weitere `mmu_i386`-Instanz, in der parallele Kontrollflüsse ablaufen können, die sich physikalisch auf einem anderen Rechner befinden. Mit ähnlichen Konfigurationen lässt sich beispielsweise verteiltes Rechnen (number-crunching) oder Prozess-Migration betreiben.

Die andere `remote`-Instanz stellt das andere Extrem von möglichen Einsatz-Szenarien dar: die bisherige „Wurzel“ der Verzeichnis-Hierarchie wird auf einem anderen Rechner verfügbar gemacht; diese Funktionalität entspricht etwa derjenigen von konventionellen Netzwerk-Dateisystemen. Wie man an diesem Beispiel sieht, braucht hierfür kein neuer Baustein-Typ implementiert zu werden. Die anderen Bausteine müssen dazu allerdings eine später genauer untersuchte Kompetenz, nämlich die *multiuser*- oder *multiversion*-Kompetenz. Nur wenn diese gegeben ist, dürfen Eingänge mehrerer Baustein-Instanzen *parallel* am

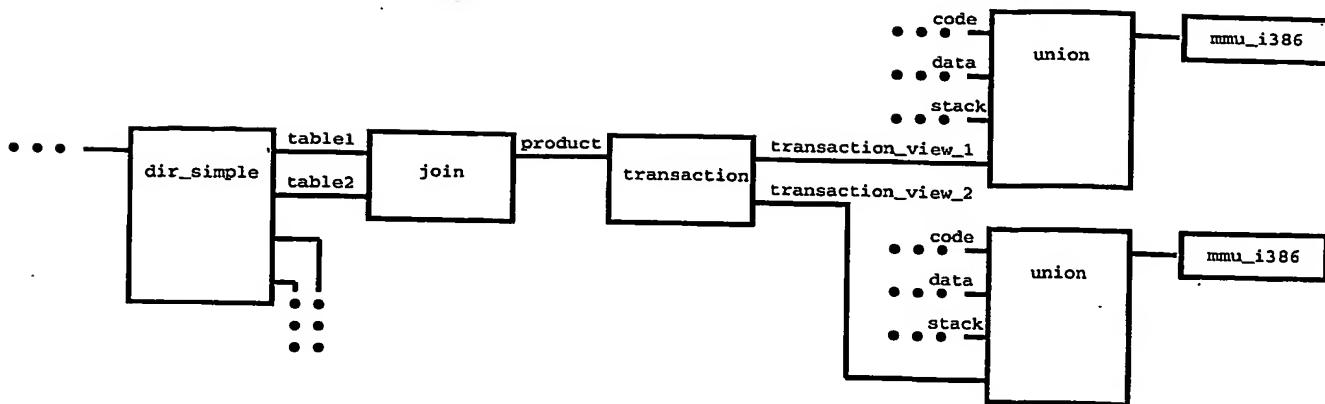


Abbildung 1.3: Drittes Beispiel

gleichen Ausgang angeschlossen werden.

Das Beispiel 1.3 soll andeuten, wie sich die Funktionalität von Datenbanken auf Basis der Abstraktionen Nest Baustein implementieren lässt. Die Grundidee besteht (relationale) Datenbank-Tabellen in Nestern abzulegen. Der Baustein *join* führt die bekannte Join-Operation durch, indem er an seinem Ausgang eine virtuelle Produkt-Tabelle zur Verfügung stellt, die lesbar und ggf. auch in gewissen Grenzen modifizierbar sein kann. Der nachfolgende *transaction*-Baustein dient als Isolations-Puffer für zwei *transaktionale Sichten*, die voneinander unabhängige *isiolierte Sichten* mit der bekannten ACID-Eigenschaft (siehe z.B. [HR83, LS87, GR93]) bereitstellt. Die beiden Sichten werden im Beispiel in Benutzer-Adressräume eingebettet; damit haben wir einen *Synergieeffekt* zwischen dem Fachgebiet der Betriebssysteme und der Datenbanken ermöglicht.

Wie man sich leicht vorstellen kann, sind transaktionale Sichten nicht nur für die klassischen Einsatzgebiete in Datenbanken nützlich, sondern können je nach Einsatz an verschiedenen Stellen einer Baustein-Hierarchie beispielsweise zur Isolation kompletter Prozessabbilder, oder im an- klem von Festplatten-Partitions-Abbildern dienen.

# Kapitel 2

## Entwurfs-Prinzipien

In diesem Kapitel möchte ich die allgemeinen Prinzipien darstellen und begründen, auf denen die vorgestellte Betriebssystem-Architektur beruht.

### 2.1 Generizität: Wahl der angemessenen Abstraktionsebene

Ausgangsbasis aller Überlegungen ist die *Funktionalität*, die ein Betriebssystem implementieren muss, wenn es bestimmte *Anforderungen* erfüllen soll.

Die Anforderungen werden in dieser Arbeit als „freie Variable“ betrachtet<sup>1</sup>. Anforderungen können sich fortlaufend ändern; daher werden *Klassen von Anforderungen* auf informelle Weise betrachtet.

Für die zu implementierende Funktionalität gilt ein informelles Gesetz, das im Volksmund als „von nichts kommt nichts“ bekannt ist. Alles, was das System leisten soll (unabhängig davon, was das sein soll), muss irgendwo irgendwie von irgendwem implementiert werden.

Gängige Methoden zur Lösung dieser Problematik werden vor allem im Software-Engineering, speziell bei den objektorientierten Entwurfs-Methoden (z.B. [Mey88]) gelehrt (Stichwort Anpassbarkeit an neue Aufgabenstellungen). Der objektorientierte Ansatz versucht dabei die *Wiederverwendbarkeit* von Software-Modulen zu verbessern, indem er die *Erweiterung* bestehender Module und Schnittstellen zum Zwecke der Anpassung an neue Aufgabenstellungen ermöglicht. In der Praxis eingesetzte objektorientierte Software mit einer längeren Entwicklungsgeschichte zeigt nicht selten eine riesige Ansammlung von Klassen mit einer ziemlich komplizierten Klassen-Hierarchie und schwer durchschaubaren Vererbungs- und Enthaltenseins-Beziehungen („is-a“ und „has-a“-Beziehungen). Bei eingehender Betrachtung findet sich darin viel *Redundanz*, die (vielleicht) hätte vermieden werden können, wenn man die jetzigen Anforderungen von Anfang an gekannt hätte und bei sorgfältigem Systementwurf berücksichtigt hätte<sup>2</sup>.

<sup>1</sup>Der oft übliche Weg besteht in der Erfassung von Anforderungen vor dem Beginn jeglicher Entwurfs-Arbeiten. Wenn dies nicht möglich ist, könnte man der Versuchung erliegen, auf Anforderungen einfach nicht zu achten. Dies kann schwerwiegende Nachteile zur Folge haben. Es geht mir daher nicht darum, eine Anforderungs-Analyse abzuschaffen, sondern diese in spätere Entwurfs-Phasen, insbesondere den Detail-Entwurf konkreter Baustein-Typen, zu verschieben.

<sup>2</sup>Dieser Effekt ist nicht auf objektorientierte Systeme beschränkt; er scheint mir ein soziologisches und psychologisches Problem bei jeglicher Software-Entwicklung mit vielen Beteiligten zu sein, und dürfte daher

Die allgemeine Fragestellung lautet daher, wie man *unnötige Redundanz* in der Implementierung möglichst weitgehend vermeiden kann, selbst wenn alle Anforderungen (noch) gar nicht bekannt sind.

#### 2.1.1 Parametrische Generizität

Ein Mittel zur Vermeidung unnötiger Redundanzen ist der Einsatz von *Generizität*.

Als Generizität betrachte ich jedes Mittel, das es erlaubt, ähnliche Dinge nur einmal hinzuschreiben, obwohl sie mehrmals vorkommen (evtl. in Varianten).

Diese Definition ist wesentlich weit-reichender als die oftmals in der Literatur anzutreffende Verwendung des Begriffs Generizität (siehe z. B. [Mey88, Kapitel 6 und 19]). Ich verwende diesen Begriff nicht in der Reinform, sondern betrachte immer nur Spielarten<sup>3</sup> von Generizität.

Eine besondere Spielart ist die *parametrische Generizität*. Beispiele hierfür sind Ersetzungsmechanismen wie z.B. der Lambda-Kalkül (siehe z.B. [Fie88]) und seine Anwendung in funktionalen Sprachen wie Lisp oder ML oder Haskell (vgl. [Tho96]), Parameter-Substitution in Makro-Präprozessoren, Parameter von Template-Mechanismen in objektorientierten Sprachen wie C++, oder Anwendungen des Schlüsselwortes *generic* in Ada. Parametrische Generizität bedeutet, dass im Unterschied zu allgemeineren Ersetzungsmechanismen wie z.B. Chomsky-Typ-0-Grammatiken oder L-Systeme keine beliebigen Ersetzungen möglich sind, sondern syntaktisch eindeutig gekennzeichnete Platzhalter, die *formale Parameter* genannt werden, durch Terme oder Ausdrücke mit einer festgelegten Syntax (*aktuelle Parameter*) substituiert werden.

In der Literatur wird häufig der nackte Begriff der Generizität in ungefähr der gleichen Bedeutung wie hier die parametrische Generizität verwendet, oder er steht in noch

auch für die hier vorgestellte Architektur ab einer gewissen Größenordnung relevant werden (Conway's Law: „systems mirror the structure of organizations producing them“). Das objektorientierte Paradigma wird jedoch manchmal so gelehrt, als wäre die Erweiterung bestehender Schnittstellen ein Allheilmittel gegen die Problematik jeglicher Veränderungen von Anforderungen. Diese oft nur unterschwellig vorhandene Einstellung führt zusammen mit der durch die objektorientierten Sprachen stark erleichterten Handhabung der Erweiterungsmechanismen sehr leicht zur Ausuferung von Klassenhierarchien (bzw. bei konventionellen Entwürfen zum Anflanschen von „Balkonen“) und damit zur vermeidbaren Reduplikation von implementierter Funktionalität.

<sup>3</sup>Über die in dieser Arbeit untersuchten Spielarten von Generizität hinaus kann es auch noch weitere geben.

weiterer Einschränkung synonym für parametrische Polymorphie von Typen (vgl. [CW85]), beispielsweise bei Meyer [Mey88, Kapitel 6 und 19]<sup>4</sup>. Das Konzept der parametrischen Generizität setzt nicht das Vorhandensein von Typprüfungen voraus<sup>5</sup>, jedoch sollten im Idealfall derartige Mechanismen vorhanden sein. Moderne funktionale Sprachen wie Haskell erlauben Typsicherheit durch automatisierte Typ-Inferenz sogar bei Typ-Parametern (parametrische Polymorphie von Typen, vgl. [CW85]). Makro-Präprozessoren führen dagegen meist nur eine reine Text-Substitution durch, die zwar universell für beliebige programmiersprachliche Konstrukte anwendbar ist, aber keine separate Prä-Compilation und keine Typprüfung von Makros ermöglicht<sup>6</sup>.

Parametrische Generizität erlaubt die Automatisierung immer wiederkehrender Programmierungs- und Formulierungs-Vorgänge. Formale Parameter können mehrmals auf verschiedene Weise durch verschiedene aktuelle Parameter substituiert werden. Dadurch ist eine kompakte Notation möglich, die Redundanz bei der Formulierung eines Algorithmus sparen hilft. Dieser Effekt ist nicht nur bei Programmierung im Kleinen, sondern auch bei der Programmierung im Großen nutzbar.

Folgenden werde ich weitere Arten von Generizität auf informelle Weise vorstellen, die sich durch parametrische Generizität darstellen und simulieren lassen. Diese informell verwendeten Begriffe stellen kein eigenständiges theoretisches Konzept dar, sondern sind vielmehr als *Denk-Kategorien* oder besser *Denk-Hilfsmittel* für den Entwurf von komplexen Systemen durch Menschen gedacht.

Die Grundregel beim Entwurf lautet: verwende Generizität, wo immer es auf einfache Weise möglich ist und wo

<sup>4</sup> Insbesondere im Vergleich zu den Möglichkeiten der Textersetzung durch Makro-Prozessoren scheint Meyers Begriff von Generizität deutlich enger gefasst zu sein als der hiesige Begriff der parametrischen Generizität.

<sup>5</sup> Meyer meint in den Kapiteln 6 und 19, dass (parametrische) Generizität ohne Typprüfung sinnlos wäre. Dem widerspricht jedoch u.a. die Lisp-Tradition seit den 1960er Jahren, in der parametrische Generizität ohne *automatisierte* Typprüfungen benutzt wurde. Man muss lediglich die Typprüfungen notfalls „von Hand“ zur Laufzeit machen, wenn man das Konzept der parametrischen Polymorphie in solchen Sprachen wie der Ur-Lisp nutzen will. Vielleicht kommt dieses Missverständnis auch daher, dass der nackte Begriff der Generizität oftmals synonym für den auf Weise eingeschränkten Begriff der parametrischen Polymorphie verwendet wird. Diese eingeschränkte Verwendung verstellt den Blick für die wesentlich weiter reichenden Möglichkeiten von Generizität im Sinne von Mechanismen zur Reduktion von Redundanz.

<sup>6</sup> Einige Beispiele sollen kurz verdeutlichen, dass Makro-Prozessoren trotz ihrer geringeren Sicherheit gegen versehentliche Fehlbenutzung einige nützliche Anwendungen von parametrischer Generizität ausführen können, die mittels parametrischer Polymorphie nicht auf einfache Weise simulierbar sind: Beispielsweise kann man mittels Makro-Prozessoren die Felder von Strukturen (Records oder Klassen-Mitglieder) alphabetisch nach dem Namen oder nach kombinierten Schlüsseln wie (Typ,Name) sortieren. Man kann die Namen der Mitglieder einem bestimmten Namens-Schema unterziehen oder die Einhaltung eines bestimmten Namens-Schemas überprüfen, beispielsweise indem die Namen aller Mitglieder einer Klasse A die Form A\_\* haben müssen. In statisch typisierten Sprachen wie Eiffel oder C++ lassen sich statisch auswertbare Iteratoren über Klassen-Mitglieder definieren, die ansonsten nur in typlosen Sprachen wie Smalltalk auf einfache Weise generisch realisierbar sind. Mittels genügend mächtiger Makro-Sprachen wie beispielsweise TeX, die vom Typ Chomsky-0 sind, kann man theoretisch sogar automatische Typ-Inferenz zwischen Klassen betreiben und abstrakte Basisklassen aus einem gegebenen Klassenschema automatisch extrahieren. Derartige Techniken könnten beispielsweise zur Entdeckung bisher unentdeckt gebliebener Redundanz in großen Software-Projekten dienen.

immer es Redundanz einsparen kann<sup>7</sup>.

## 2.1.2 Erweiterungs-Generizität

Die in der Objektorientierung verwendete Erweiterung bzw. Spezialisierung von Schnittstellen durch Vererbung (gelegentlich auch die Einschränkung von Schnittstellen) lässt sich ebenfalls als eine besondere Art bzw. als Spezialfall von parametrischer Generizität charakterisieren: *Erweiterungs-Generizität*. Erweiterungs-Generizität lässt sich auch durch theoretische Begriffe wie Untertypen (subtyping, vgl. [CW85]) oder praxisnähere Ausprägungen wie Abstrakte Basisklassen beschreiben.

Im Anhang A wird durch ein Beispiel nachgewiesen, dass sich die Vererbung prinzipiell durch parametrische Generizität simulieren lässt<sup>8</sup>. Dieses Beispiel soll kein Vorbild guten Software-Engineerings darstellen, sondern lediglich zeigen, dass man *prinzipiell* die Vererbung sogar mit primitiven Mitteln wie dem C-Präprozessor nachbilden kann, obwohl der C-Präprozessor nicht einmal bedingte Ausdrücke *innerhalb* von Makro-Expansionen auswerten kann (was jedoch zur Nachbildung überschriebener Methoden sehr nützlich wäre). Der interessierte Leser, der das Beispiel aus Anhang A nachzuvollziehen versucht, wird auf den Kunstgriff verwiesen, dass man in einigen Makro-Sprachen die Namen von zu expandierenden Makros auch berechnen kann; im C-Präprozessor benötigt man dazu den Konkatenations-Operator ##. Dieser Kunstgriff wird u.a. auch gerne in Kreisen von TeX-Programmierern eingesetzt (vgl. \csname in [Knu94]).

Aus theoretischer Sicht handelt es sich hierbei um Funktionen höherer Ordnungen. Einige Verfechter objektorientierter Entwurfsmethoden wie z.B. [Mey88, Abschnitt 3.6.3] benutzen im Vergleich zu hier einen eingeschränkten Begriff von Generizität, dem sie dann nicht die Mächtigkeit zutrauen, den er bei entsprechender Fassung in der Theorie jedoch besitzt. Diese Mächtigkeit umfasst auch wesentliche Grundkonzepte der Objektorientierung. Daher betrachte ich Generizität schlechthin als Oberbegriff, unter den die Objektorientierung als Spezialfall untergeordnet wird.

## 2.1.3 Kompositorische Generizität

Eine weitere besondere Form von Generizität spielt bei uniformen IO-Schnittstellen wie [Che87], in stapelbaren

<sup>7</sup> Übermäßiger Einsatz von Generizität kann auch die Redundanz erhöhen, die Durchschaubarkeit von Software verschlechtern, die Performanz verschlechtern, und damit zu negativen Effekten führen. Eine objektiv exakte Einschätzung und Abwägung der Einsatzstellen scheint schwierig und ist daher oftmals der subjektiven Einschätzung des jeweiligen Entwerfers unterworfen.

<sup>8</sup> Dieses Beispiel widerspricht nicht der scheinbar genau umgekehrten Aussage von Meyers Analyse in [Mey88, Abschnitt 19.3]. Meyer verwendet den Begriff der Generizität synonym zu einer Ausprägung von parametrischer Polymorphie, wie sie speziell in Ada auftritt. Im Gegensatz dazu verstehe ich unter parametrischer Generizität ein sehr viel allgemeineres Konzept, unter Generizität schlechthin sogar eine nochmals allgemeinere Denk-Kategorie. Man muss sich diese unterschiedlichen Begriffsbildungen vor Augen halten, wenn man die Aussagen miteinander vergleicht. Das hier vorgestellte Beispiel stützt jedoch die hier vorgeschlagene Begriffsbildung.

Dateisystem-Layern wie [HP94]<sup>9</sup>, in den Streams von System V Release 4 [GC94, Kapitel 7], oder beim Port-Konzept von Gnu Hurd [Hur] eine Rolle, die man als Vorfänger der hier vorgestellten Architektur ansehen kann. Konzeptuell wird bei den genannten Systemen die gleiche Art von Generizität wie bei der vorgestellten Architektur favorisiert, die ich *kompositorische Generizität* nennen möchte.

Auf den ersten Blick scheint kompositorische Generizität keine parametrische Generizität zu sein, da die syntaktische Darstellung oftmals anders als in funktionalen Sprachen oder Makro-Sprachen aussieht. Wie wir jedoch in Abschnitt 6.3 sehen werden, implementieren Bausteine Funktionen in einem weiteren Sinn, wobei die Verdrahtung von Bausteinen zu komplexeren Netzwerken sich als eine spezielle Art der Komposition von Funktionen auffassen lässt. Kompositorische Generizität ist ebenfalls eine Variante von parametrischer Generizität, die sich allerdings von der Erweiterungs-Generizität in der Entwurfs-Philosophie (nicht notwendigerweise jedoch in den verwendeten Mechanismen) stark unterscheidet: während objektorientierte Paradigmen ebenso wie die in den 1970er Jahren populären Hierarchie-Konzepte [Dij68, Dij71, BS75] ausdrücklich die Erweiterung von Schnittstellen bzw. von abstrakten Maschinen um neue Funktionalität durch Hinzunahme weiterer Operationen und Abstraktionen beziehen und anstreben, wird bei kompositorischer Generizität das Gegenteil angestrebt: die Schnittstelle muss in den Grundzügen auf allen oder möglichst vielen Hierarchieebenen gleich oder zumindest weitgehend kompatibel bleiben, damit eine vielfältige Kombinierbarkeit möglich ist. Bei kompositorischer Generizität wird die Erweiterung der Funktionalität nicht durch Erweiterung der Schnittstellen, sondern durch Hinzunahme weiterer Funktionalität *innerhalb* der kombinierten Funktionen erzielt; neue Funktionalität ergibt sich mithin durch die *besondere Art der Verdrahtung zu einem Netzwerk*.

#### 2.1.4 Universelle Generizität

Um kompositorische Generizität über das bisherige Anwendungsfeld von IO-Schnittstellen [Che87] hinaus auf das gesamte Betriebssystem ausdehnen zu können, müssen nicht nur die Urbild- und Bildbereiche der beteiligten Funktionen zueinander passen, sondern diese müssen darüber hinaus eine weitere Art von Generizität ermöglichen, die ich *universelle Generizität*<sup>10</sup> nennen möchte. Beispiele für universelle Konstrukte sind Turingmaschinen oder Registermaschinen, die prinzipiell andere Turing- oder Registermaschinen *simulieren* können. Universelle Generizität ist analog dazu die Fähigkeit, alternative Funktionalitäten, Repräsentationen, Zugriffsverhalten, Granularitäten etc. auf relativ ein-

<sup>9</sup>Die in [HP94] vorgestellte hierarchische Modularisierung von Dateisystemen verwendet einige der hier vorgestellten Konzepte, beschränkt sich jedoch nicht nur auf die Funktionalität von Dateisystemen und verwendet deren aufwendigere Schnittstellen (insbesondere zum Management von Dateisystem-Unterbäumen), sondern basiert auch inhärent auf konventionellen Dateisystem-Konzepten und -Abstraktionen wie vnodes, vnode-Operationen und vFs, die in der vorliegenden Arbeit keine Rolle spielen.

<sup>10</sup>Auch universelle Generizität lässt sich durch parametrische Generizität ausdrücken. Die Unterscheidung der verschiedenen Arten von Generizität ist daher letztlich nur ein menschlicher Verstehensansatz, der eine bestimmte *Denkweise* beim Entwurf *kommunizierbar* machen soll.

fache Weise<sup>11</sup> simulieren zu können; der Begriff ist daher kein absoluter Begriff, sondern gilt immer nur relativ zum Universum der jeweils simulierbaren alternativen Konzepte.

Ein Beispiel für universelle Generizität in Betriebssystemen ist die bekannte File-IO-Schnittstelle von Unix [RT74], die aufgrund ihrer Fähigkeit zu IO mit beliebiger dynamischer Blocklänge und beliebigen Zugriffsmustern auch ältere File-Konzepte wie z.B. Lochkartenstapel mit fester Satzlänge simulieren kann, darüber hinaus aber auch unterschiedliche Zugriffsmuster verschiedener Leser und Schreiber ermöglicht, die mit den vorher üblichen satzorientierten IO-Schnittstellen nicht auf einfache Weise realisierbar waren. Dieses inzwischen zur Selbstverständlichkeit gewordene historische Beispiel einer Vereinfachung durch universelle Generizität zeigt auf, dass „weniger oft mehr“ ist: der Verzicht auf das Konzept eines „Datensatzes“ (record) in Unix macht nicht nur die Schnittstellen und die Implementierung einfacher, sondern verbessert darüber hinaus sogar noch die Funktionalität. Universelle Generizität ist daher in hohem Maße erstrebenswert und wird insbesondere beim Entwurf der Nest-Abstraktion (Kapitel 3) eine herausragende Rolle spielen.

#### 2.1.5 Fragen der Anwendung von Generizität

Die zentralen Fragen bei der Anwendung von kompositorischer und universeller Generizität lauten:

- Wie sollen die Bild- und Urbildbereiche aussehen?
- Welche Funktionalitäten sollen durch die Bausteine realisiert werden?

Diesen Fragen liegt das gemeinsame Problem der *Wahl der am besten geeigneten Abstraktionsebene* für die zu lösenden Aufgaben (bei teilweise unbekannten Anforderungen) zu Grunde. Die Vorstellung und Propagierung eines bestimmten Mechanismus alleine nicht viel über deren konkrete Anwendung in einem bestimmten Gebiet aus. Entwurfs-Entscheidungen sind nicht immer eindeutig aufgrund gesicherter Methoden und Erkenntnisse lösbar, da bereits die Betrachtung der Problemstellung ein menschlicher Interpretationsprozess ist. In dieser Arbeit werde ich daher der *Pragmatik* einen vergleichsweise großen Raum einräumen und syntaktische und semantische Konstrukte nur zur Verdeutlichung und Notation der zugrundeliegenden Ideen verwenden.

Eine konkrete Wahl von Abstraktionsebenen könnte die berechtigte Kritik auf sich ziehen, daß präzise Aussagen über Vor- und Nachteile bestimmter Entwurfs-Entscheidungen nicht oder nicht ausreichend möglich sind.

<sup>11</sup>Die Problematik bei einer formaleren Fassung des Begriffes der universellen Generizität besteht darin, was man unter *einfach* verstehen soll. Lässt man „komplizierte“ Simulationen zu, dann kann man wesentliche Teile der Funktionalität in der Abbildung auf das zu simulierende Konzept verstecken, anstatt es im benutzten Grundkonzept zu lösen. Die „Einfachheit“ einer Simulation lässt sich auf mehrere verschiedene Weisen formal fassen; da es mir bei dieser Diskussion nur um Denk-Kategorien während der Entwurfs-Phase geht, habe ich auf formale Fassungen all dieser Begriffe verzichtet.

- Im Forschungsgebiet der Betriebssysteme hat es bisher meines Wissens nach keine Evaluation verschiedener Entwurfs-Konzepte auf der Ebene ganzer Betriebssysteme gegeben, die diese Konzepte *als solche* auf einer wirklich *vergleichbaren* Grundlage (d.h. deren *isolierte* Auswirkung auf gut messbare Größen, wie Performanz oder weniger gut messbare Größen wie die Änderungsfreundlichkeit) *ohne Einfluss weiterer versteckter Parameter* wie z.B. die Kenntnisse und Fähigkeiten der jeweiligen Implementierer untersucht hätten<sup>12</sup>, da solche Evaluationen nach aktuellem Stand kaum möglich erscheinen, ist es stattdessen üblich, das Für und Wider verschiedener Entwurfs-Alternativen durch informelle Argumentation darzustellen.

Der Begriff der „Architektur“ wird in dieser Arbeit in einem weiteren Sinn verwendet als vielerorts in der Literatur (vgl. z.B. [PC75, Jon80, MP81, B<sup>+</sup>95, Lie95b, EKO95, Ass96])<sup>13</sup>, da er weder feste Implementierungsparadigmen wie „communicating sequential processes“ (siehe [Hoa78]) vorschreibt<sup>14</sup>, noch die Frage nach Kern-Größen und -Umfängen (vgl. [Lie95b]) in festliegender Weise beantwortet, noch die Frage nach den Grenzen zwischen Benutzerräumen und Kern- oder Server-Adressräumen feststellt. Wenden diese Fragen je nach Systemkonfiguration an, so müssen kann, ohne den Quelltext des Systems ändern zu müssen. Im Vergleich zur konventionellen Handhabung des Architektur-Begriffes könnte man die hier vorgestellte Architektur daher auch als Meta-Architektur bezeichnen; ich habe davon jedoch Abstand genommen, da der Begriff „Architektur“ laut Brockhaus schlicht „Baukunst“ bedeutet, und daher die in der Literatur teilweise vorkommende engeren Auffassung nicht impliziert. In der Bedeutung als „Meta-Architektur“ werden einige konventionelle Architekturfragen, die bisher als „festverdrahtet“ betrachtet worden sind und deren Diskussion in der Literatur große Beachtung gefunden hat, zu *dynamischen Parametern*, deren Variation einen direkten Vergleich dieser Paradigmen bei einer Evaluation erleichtert<sup>15</sup>.

<sup>12</sup> Ausnahmen hiervon gibt es nur bei Untersuchungen über konkrete Alternativ-Mechanismen bei speziellen wohldefinierten Teilproblemen.

Um anscheinend wenig beachteten Artikel [Ant90] beschreibt Antonov in teilweise sehr schwer zu verstehendem und manchmal unlogisch zu interpretierendem Englisch, eine „reguläre Architektur“, die aufz. anderer formeller Darstellung dem Kerngedanken der hier vorgestellten Architektur sehr nahe kommt, dabei jedoch auf Erweiterungs-Generizität und weniger auf kombinatorische Generizität ausgelegt ist. Im Detail gibt es weitere gravierende Unterschiede, so etwa bei der enormen Anzahl von Schnittstellen-Funktionen, bei den Instantiierungs-Mechanismen, den fest eingebauten Schutzmechanismen, und möglicherweise auch bei der Anzahl der „Ausgänge“ pro „Baustein“ (in allen Beispielen kommt immer nur ein einziger Ausgang vor; einen grammatischen Hinweis auf mehrere „Ausgänge“ konnte ich nicht finden). Weiterhin fehlt bei Antonov der hier vorgestellte Ersatz von Dateisystemen durch dynamische hierarchische Instantiierung von Bausteinen; eine Unterscheidung verschiedener Betriebsarten oder Betriebsmodelle ist ebenfalls nicht zu erkennen.

<sup>14</sup> Eine Ausnahme ist [Str78], wo Stroustrup die Unabhängigkeit und Austauschbarkeit der Implementierungsparadigmen von den Modulschnittstellen aufzeigt, und damit den Architekturbegriff ebenfalls weiter auffasst. Leider scheint dieser grundlegende Artikel wenig Beachtung in der Betriebssystem-Literatur gefunden zu haben, da auch Jahrzehnte später die Verwendung fester Implementierungsparadigmen immer noch zum Stand der Technik zu gehören scheint.

<sup>15</sup> Dies gilt nur für Entwurfs-Entscheidungen, die *innerhalb* der hier vorgestellten Architektur offengelassen und daher parametrisierbar sind; auf der Ebene der Meta-Architektur selbst werden jedoch auch von mir feste

Zur Begriffsbildung: statt mit Architektur oder Meta-Architektur könnte man die vorliegende Entwurfs-Methodik auch als *Framework*<sup>16</sup> oder als *Komponenten-Architektur*<sup>17</sup> bezeichnen. Da diese Begriffe nach weit verbreiteter Auffassung zur Zeit sehr eng mit der Objektorientierung verknüpft sind, die ich in der vorliegenden Arbeit zwar ermögliche, aber nicht zur zwingenden Voraussetzung der Architektur erhebe, habe ich von einer derartigen Begriffsbildung vorerst Abstand genommen. Im Unterschied zu Szyperskis Komponenten-Begriff [Szy98] könnte man die hier vorgeschlagenen Bausteine auch als *leichtgewichtige*<sup>18</sup> *instantiierbare*<sup>19</sup> *Komponenten* charakterisieren.

Bei Konflikten zwischen verschiedenen Arten von Generizität bei zu treffenden Entwurfs-Entscheidungen propagiere ich zumindest für die Anwendung in Betriebssystemen die folgende Priorisierung miteinander konkurrierender Arten von Generizität:

Entscheidungen gefällt, die prinzipiell (wie auch bei bisherigen Architekturen) der oben genannten Kritik unterliegen und im Einzelfall weiterer Evaluation und ggf. einer Revision bedürfen.

<sup>16</sup> Dieser Begriff wird in der Praxis meist mit der Objektorientierung verknüpft (vgl. z.B. [JH02, S. 273]: „Frameworks stellen komplexe, partielle Programme dar, deren Instantiierung zu einer Applikation ohne Vererbung nicht denkbar wäre“), bis hin zur Bezeichnung einer *konkreten Implementierung* einer Sammlung von abstrakten Basisklassen, und könnte daher zu Missverständnissen führen. Eigentlich würde der Begriff Framework gut auf die hier vorgestellte Architektur passen, wenn er nicht bereits auf die zitierte Art eingeschränkt wäre. Eventuell wäre es sinnvoll, über eine Erweiterung dieses Begriffs nachzudenken; darüber sollte jedoch ein weitgehender Konsens erzielt werden.

<sup>17</sup> Zum aktuellen Diskurs zwischen Broy / Sindersleben [BS02] und Jähnichen / Herrman [JH02] über Wesen, Vor- und Nachteile der Objektorientierung, insbesondere zum Zusammenhang zwischen Komponenten und Objektorientierung, kann die vorliegende Arbeit eine Erweiterung der Perspektive beitragen. Man könnte die hier vorgestellte Architektur ohne weiteres als Komponenten-Architektur bezeichnen, obwohl sie nicht *unbedingt* auf der Objektorientierung aufbaut und damit von der vorherrschenden Meinung in dieser Hinsicht abweicht. Wie in Abschnitt 6.2 gezeigt wird, wird objektorientierte Vererbung von der hier vorgestellten Architektur *ermöglicht*, aber nicht zur zwingenden *Voraussetzung* erhoben. Ich habe in den Beispielen des Kapitels 4 besonderen Wert darauf gelegt, die Grundfunktionalität eines Betriebssystems möglichst nur mit Hilfe kompositorischer Generizität zu realisieren und Erweiterungs-Generizität praktisch vollständig zu vermeiden. Damit existiert ein Beispiel, das eine Komponenten-Architektur ohne feste Verknüpfung mit einer wesentlichen Grundidee der Objektorientierung (egal ob man sie nun mit „Vererbung“ oder als Spezialfall einer „Delegation“ bezeichnet) realisiert. Damit dürfte genügend Substanz zur Erhöhung der These vorgebracht worden sein, dass Komponenten-Architekturen als weitgehend orthogonal zur Anwendung spezifisch objektorientierter Entwurfs-Methoden betrachtet werden können. Eine solche Betrachtung hängt allerdings von der Definition von „Objektorientierung“ ab; je nach Geschmack oder Herkunft aus einer bestimmten Schule kann man die jeweiligen Begriffe so definieren, dass Objektorientierung einen Oberbegriff darstellt, unter dem auch Komponenten-Architekturen als ergänzende Anwendung des Prinzips der kompositorischen Generizität fallen, oder genau umgekehrt, oder dass zwei zueinander orthogonale Begriffe entstehen. Ich plädiere aus Gründen der methodischen Sparsamkeit für die dritte Variante.

<sup>18</sup> Vgl. [Szy98, Abschnitt 4.1.8]. Das vermutete Phänomen „maximizing reuse minimizes use“ scheint mir auf der impliziten Annahme von Erweiterungs-Generizität als Maximierungs-Strategie über Komponenten-Grenzen hinweg zu beruhen; dieses Phänomen wird durch das Konzept der kompositorischen Generizität und der universellen Generizität in sein Gegenteil verkehrt.

<sup>19</sup> Szyperski [Szy98] betont mehrmals (Abschnitte 1.5, 4.1.1), dass die Instantiierung von Komponenten wegen der Statuslosigkeit sinnlos wäre. Im Unterschied dazu haben beispielsweise mehrfache Instanzen von *dir\_\**-Bausteinen durchaus Sinn, da es durchaus auf die *Stellung* in einer Baustein-Hierarchie bzw. auf den *Verdrahtungs-Kontext* ankommt.

1. Universelle Generizität
2. Kompositorische Generizität
3. Erweiterungs-Generizität

Eine genaue Evaluation der Vor- und Nachteile dieser Priorisierung für die Anwendung in Betriebssystemen steht noch aus, mindestens solange das hier vorstellte Konzept noch nicht in die Praxis umgesetzt worden ist.

## 2.2 Trennung von Mechanismus / Strategie / Repräsentation

Im Betriebssystem-Bau ist das Prinzip der Trennung von *Mechanismus* (mechanism) und *Strategie* (policy) schon sehr lange bekannt und benutzt worden; eine Darstellung dieses Prinzips und ihrer Vorteile findet man z.B. in [LCC<sup>+</sup>75, CJ75].

Als unabhängig von Mechanismen und Strategien ist fernerhin die *Repräsentation* zu betrachten<sup>20</sup>. Beispielsweise lässt sich ein Monitor [Han73, Hoa74] als programmiersprachliche Repräsentation eines Mechanismus verstehen, der prinzipiell wie ein binäres Semafor funktioniert; die programmiersprachliche Repräsentation eines Monitors lässt sich bekanntermaßen auf einfache Weise mittels Semaforen und konventionellen Sprachmitteln simulieren<sup>21</sup>. Hieraus ist zu erkennen, dass die Auswahl einer einzigen Repräsentation ausreichend ist und Redundanz vermeiden hilft<sup>22</sup>; diese Repräsentation sollte jedoch möglichst universell und einfach handhabbar / verstehtbar sein. Die Entscheidung für die Auswahl einer bestimmten syntaktischen Repräsentation ist daher der Pragmatik zuzurechnen; im Rahmen dieser Arbeit wird nicht weiter darauf eingegangen; insbesondere werden Repräsentationsfragen nicht zur Auswahl von Mechanismen oder Strategien herangezogen.

Die Abgrenzung zwischen Mechanismus und Strategie ist nicht immer eindeutig möglich, da sie vom Kontext und der gewählten Abstraktionsebene abhängt. Eine Strategie kann beispielsweise ihrerseits wieder in Unter-Strategien und Unter-Mechanismen zerfallen. Auch Mechanismen können in ihrer Feinstruktur wiederum Strategien enthalten.

<sup>20</sup>Eine ähnliche Auffassung vertritt Parnas in [Par78].

<sup>21</sup>Hiervon unabhängig sind wiederum die Warte-Strategien mehrerer eintrittswilliger Prozesse wie beispielsweise die FIFO-Strategie (als interessantes Realisierungsbeispiel hierzu siehe [RK79]).

<sup>22</sup>In der Praxis ist dieser Vorsatz nicht immer konsistent durchzuhalten, vor allem in größeren Projekten mit vielen Beteiligten, die unterschiedliche Interessen verfolgen. So könnte es beispielsweise vorkommen, dass eine Interessengruppe die Programmiersprache Ada bevorzugt, die andere die Sprache C (Argumente für die jeweiligen Positionen sind sattsam bekannt: Ohne Zweifel ist Ada die sicherere, produktivere und wartungssärmere Sprache, der Großteil bereits existierender und ausgetesteter Software ist jedoch in C oder C++ geschrieben, zu der unbedingt Kompatibilität zu wahren ist). Prinzipiell lässt sich die hier vorgestellte Architektur nicht nur einheitlich in einer der beiden Sprachen realisieren, sondern auch bausteineweise in beliebigen Mischungen, sofern die Aufruf-Konventionen der jeweiligen Sprachen und Compiler zueinander kompatibel sind. Da dies nicht nur die Redundanz, sondern auch den Integrationsaufwand und mögliche Fehlerquellen erhöhen kann, sollte dies möglichst vermieden, auf jeden Fall aber sehr gut überlegt und begründet werden.

Auf relativ grober Abstraktionsebene korrespondieren Mechanismen und Strategien mit den hier propagierten Abstraktionen Nest und Baustein auf ziemlich direkte Weise: ein Nest stellt einen Satz von abstrakten Mechanismen zur Verfügung, ein Baustein implementiert eine Strategie oder eine Menge von Strategien.

Die Nest-Schnittstelle stellt insofern *abstrakte* Mechanismen bereit, als dass die konkrete Implementierung der Mechanismen durch späte Bindung (late binding) zu Stande kommen kann.

Die Trennung zwischen Mechanismus und Strategie wird durch die hier vorgestellte Architektur nicht nur direkt unterstützt, sondern geradezu zum Prinzip erhoben. Da Bausteine wiederum andere Bausteine enthalten oder über ihre Eingänge benutzen können, lässt sich das Prinzip der *schrittweisen Verfeinerung* auf die Trennung in Mechanismen und Strategien anwenden: eine Baustein-Hierarchie fungiert gleichzeitig als hierarchisch-rekursive Zerlegung des zu lösenden Problems in Mechanismen und Strategien.

## 2.3 Das Verantwortungs-Prinzip

*Verantwortung* und ihre Aufteilung ist ein Prinzip, das in komplexeren menschlichen Gesellschaften und Systemen wie Firmen oder Behörden seit Jahrtausenden mit Erfolg praktiziert wird; als Ergebnis eines Aufteilungsvorgangs von Verantwortung entstehen *Zuständigkeiten*. Im Kontext von Betriebssystemen wurde Verantwortung ebenfalls als Strukturierungs-Leitlinie eingesetzt (ohne dass dies den jeweiligen Autoren völlig bewusst gewesen sein muss<sup>23</sup>): die in den 1970er Jahren populären hierarchischen Strukturen [Dij68, Dij71] lassen sich beispielsweise durch Aufteilung von Verantwortung gewinnen, ebenso die Weiterführung dieser Idee in sogenannten „objektorientierten“ Betriebssystemen (beispielsweise [K<sup>+</sup>81, Y<sup>+</sup>90, B<sup>+</sup>95, Ass96] u.v.m.), aber auch die auf bestimmten festen Mechanismen aufgebauten Betriebssysteme (z.B. auf Capabilities basierend [NW77, T<sup>+</sup>90]). Auch das im Software-Engineering oft zitierte Lokalitäts-Prinzip als Mittel zur Strukturierung von Systemen beruht letztlich auf der Aufteilung von Verantwortung. Es gibt unzählige weitere Beispiele.

### 2.3.1 Kontrollflüsse

Ein *sequentieller Kontrollfluss* (thread, konzeptuelle Beschreibung bereits in [DH66]) lässt sich als Aktivitätsträger charakterisieren, der von außen beobachtbar nur eine sequentielle Folge von (Maschinen-)Operationen ausführt. Kontrollflüsse sind bereits in [Dij68] zur Strukturierung von Verantwortung in Betriebssystemen benutzt worden; diese

<sup>23</sup>Wie [Mey88] und viele andere Autoren bemerken, verlaufen die Haupt-Auseinandersetzungen um Strukturierungs-Kriterien hauptsächlich entlang der Demarkations-Linien „Strukturierung nach Daten“ versus „Strukturierung nach Kontrollfluss“ (vgl. [Par72]). Die von mir propagierte Strukturierung nach *Verantwortungsbereichen* nimmt demgegenüber eine übergeordnete Perspektive ein: Verantwortung ist fast immer an Datenstrukturierung und nur selten an Kontrollflüsse gekoppelt. Die Abstraktionsebene ist jedoch von konkreten Datenstrukturen abgekoppelt: bei hoher Abstraktionsebene kann sich Verantwortung auf *Mengen* von Datenstrukturen erstrecken, im anderen Extrem aber auch bis auf Teil-Datenstrukturen verfeinert werden.

feste Verbindung von Kontrollflüssen mit der Verantwortung für bestimmte Bereiche wird in der vorliegenden Arbeit jedoch aufgegeben. Kontrollflüsse können zur Strukturierung von Verantwortung benutzt werden, jedoch gibt es in der hier vorgestellten Architektur keine *notwendigerweise feste Verbindung* mit Verantwortungsbereichen (wobei im Einzelfall auch feste Verbindungen hergestellt werden können, aber nicht müssen).

Kontrollflüsse werden als vollkommen unabhängig von konventionellen Konzepten wie *Prozesse* oder Mechanismen wie *Schutzbereiche* angesehen. Im Gegensatz zu anderen Arbeiten werde ich den Begriff des Prozesses vermeiden, da es für ihn mehrere verschiedene Definitionen gibt<sup>24</sup>. Seit der Wiederentdeckung der Schutzbereiche (spheres of protection [DH66], protection domains [NW74, CJ75]) im Kontext so genannter Single-Address-Space-Systeme [C<sup>+</sup>94, Ros94] (vgl. Beschreibung und Schutzmechanismen von CTSS in [BPS81]) hat sich die Bedeutung der Begriffe weiter verkompliziert. Ich beschränke mich daher auf die Begriffe Kontrollfluss und Schutzbereich.

Jeder Kontrollfluss hat zu jedem Zeitpunkt einen einzigen bestimmten Kontext. Die Bedeutung dieses Kontexts hängt vom jeweils gewählten *Ausführungs-Modell* ab: bei der Simulation eines „Prozesses“ wäre dies beispielsweise das Prozessabbild, in dem sich der Kontrollfluss befindet und seine Operationen ausführt; im Fall von Single-Space- oder Hybridsystemen wäre dies der jeweils aktuelle Schutzbereich.

Kontexte können auf Anforderung *gewechselt* werden (vgl. [DH66]): ein Kontrollfluss wechselt damit in einen anderen „Prozess“ bzw. in einen anderen Schutzbereich, der sich ggf. auch auf einem anderen Rechner befinden kann. Hierzu sind zwei verschiedene Varianten denkbar: eine ohne Abspeichern des alten Kontextes im Stile von Wechseln zwischen Coroutinen, die andere mit Abspeichern aller durchlaufenden Kontexte in Form eines Stapelspeichers mit der Möglichkeit zur Rückkehr in einen früheren Kontext ähnlich einem synchronen RPC oder LRPC. Eine Konsequenz aus der Wechselmöglichkeit ist, dass die Anzahl der in einem „Prozess“ oder Schutzbereich tätigen Kontrollflüsse dynamisch ändern und auch zeitweise zu Null werden kann.

Man sollte sich vor Augen halten, dass Schutzbereiche einen Mechanismus zur Durchführung und Überwachung von Sicherheits-Konzepten darstellen, wohingegen Verantwortungsbereiche ein davon prinzipiell unabhängiges *logisches Konzept* zur Strukturierung eines Systems darstellen. Die Abbildung von Verantwortungsbereichen auf Schutzbereiche ist eine Frage von Strategien und kann auf unterschiedliche Weise, ggf. auch dynamisch zur Laufzeit, gelöst werden.

Kontrollflüsse werden in konventionellen Architekturen als eigenständige Abstraktion betrachtet. Da ihre Implementierung in einem eigenen Baustein erfolgen kann, ist die Behandlung als separate Abstraktion nicht unbedingt not-

wendig. Wegen ihrer dynamischen Durchdringung der gesamten Infrastruktur kann man sie jedoch als *Hilfsabstraktion* bezeichnen.

Im Folgenden betrachten wir unser Betriebssystem zur Vereinfachung standardmäßig (sofern nicht ausdrücklich abgewichen wird) weder mit Hilfe von Prozess- oder Schutzbereichen, noch von Kontrollflüssen oder anderen Aktivitätsträgern, sondern ausschließlich auf Basis der jeweils zu implementierenden und zu verantwortenden Funktionalität.

### 2.3.2 Schnittstellen-Mechanismen

Die Gesamtverantwortung eines Betriebssystems kann auf sehr viele verschiedene Arten aufgeteilt werden; die von mir bevorzugten Aufteilungs-Strategien werden in Abschnitt 2.4 näher dargestellt. Jegliche Art von Aufteilung von Verantwortung führt dazu, dass *Modularisierungsgrenzen* in ein System eingeführt werden, die bei formalisierter Darstellung auch als *Schnittstellen-Instanzen* bezeichnet werden.

Durch die Einführung von Schnittstellen-Instanzen zur Begrenzung von Modul-Instanzen entsteht eine Unterscheidung der durchzuführenden *Operationen* in *Aufrufer*- und *Bearbeiter*-Instanzen, die jeweils immer nur relativ zu einer festen Schnittstelle so bezeichnet werden; dieselbe Instanz kann bezüglich verschiedener Schnittstellen gleichzeitig als Aufrufer oder Bearbeiter fungieren. Bei verschiedenen unterscheidbaren Operationen können diese Rollen unterschiedlich verteilt sein, beispielsweise kann eine Modul-Instanz *A* gegenüber der Instanz *B* bei der Durchführung der Operation *op*<sub>1</sub> als Aufrufer, bei *op*<sub>2</sub> als Bearbeiter fungieren.

Als Mechanismen zur Weitergabe der Verantwortung einer Aufrufer- an eine Bearbeiter-Instanz kommen mehrere bekannte und bewährte Methoden in Betracht. Beispielsweise kommen hierfür in Frage, in der Reihenfolge fallenden Overheads bzw. fallender Kosten:

- RPC (remote procedure call) [BN84, TA90]
- LRPC (local / lightweight RPC) [B<sup>+</sup>90]
- Indirekte Prozeduraufrufe
- Direkte Prozeduraufrufe
- Makro- oder Inline-Prozeduraufrufe

Diese Beispiel-Mechanismen eignen sich teilweise nur für die *synchrone* Weitergabe von Verantwortung, bei der die Aufrufer-Instanz bis zur Beendigung der Bearbeitung *warten* muss. Bei *asynchroner* Weitergabe der Verantwortung entsteht implizit ein weiterer *logischer Kontrollfluss*<sup>25</sup>. Die

<sup>24</sup>Heute wird unter Prozess meist ein in Ausführung befindliches Programm verstanden, das eine Menge von Ressourcen allokiert hat und mindestens einen Kontrollfluss (thread) besitzt. In den 1960er und 1970er Jahren waren teilweise noch einfachere Definitionen üblich, die mit dem heutigen Begriff des Kontrollflusses (thread) zusammenfallen.

<sup>25</sup>Es muß nicht unbedingt ein tatsächlich neuer Kontrollfluss im physischen Sinne entstehen: Bei der Implementierung von Bausteinen als Server-Prozesse (vgl. [Han70, Hoa78]) bleibt die Anzahl physischer Kontrollflüsse im Regelfall konstant. Die im Nachrichtensystem gepufferte Nachricht stellt im logischen Sinne einen logischen Kontrollfluss dar, der von einem physischen Kontrollfluss simuliert wird, sobald die Nachricht bearbeitet wird. Solche Systeme sind für den Fall gebaut, dass mehr logische als physische Kontrollflüsse auftreten können, und beschränken daher die theoretisch mögliche maximale Parallelität auf künstliche Weise. Falls

asynchrone Weitergabe von Verantwortung per Prozeduraufruf ist daher nicht ohne Einbezug des jeweils verwendeten Kontrollfluss-Konzeptes (siehe Abschnitt 2.3.1) möglich.

Man sollte sich vor Augen halten, dass die Frage nach einer parallel oder sequentiell übertragenen Verantwortung prinzipiell von der Aufteilungsstrategie in Module unabhängig ist und daher als eine Frage der konkreten Implementierungsstrategie der Module gesehen werden kann, auch wenn die zugehörigen Mechanismen nicht vollkommen (wenn auch weitgehend) voneinander unabhängig sind.

Universelle Generizität relativ zu den oben genannten synchronen Mechanismen lässt sich direkt durch einheitliche Verwendung von synchronem RPC auf allen Ebenen des Systems erzielen, da dieser die Semantik der anderen Mechanismen als Spezialfall mit umfasst. Dies würde jedoch die Performanz eines lokalen Rechners in unakzeptabler Weise verschlechtern. Zur Erzielung universeller Generizität schlage ich daher eine Erweiterung der bereits in [Str78] dargestellten Methode vor:

Man wähle eine einheitliche und bequeme syntaktische Repräsentation, beispielsweise die Prozeduraufruf-Syntax der verwendeten Programmiersprache. Die *Bindung* von Instanzen dieser Konstrukte an einen der obigen Mechanismen erfolgt dann entweder zur Übersetzungszeit des jeweiligen Moduls, oder falls möglich auch zur Laufzeit bei der Instantiierung eines Moduls (die Unterscheidung zwischen indirekten / direkten / Inline-Aufrufen ist mit heute gängiger Standard-Technik<sup>26</sup> nicht ohne weiteres zur Laufzeit möglich).

### 2.3.3 Abgrenzung von Verantwortung durch Kapselung

Im Software-Engineering ist lange bekannt, dass die Prüfbarkeit, Wartbarkeit und viele andere Eigenschaften von Modulen verbessert werden, wenn die Schnittstellen möglichst „dünn“ sind und möglichst wenig Information über die innere Struktur der Implementierung preisgeben (Prinzip der *Verbergung von Information*, vgl. [Par72]).

Dies bedeutet zum einen, dass Schnittstellen stets offen gelegt werden müssen, wobei die Schnittstelle zwischen Bausteinen eine von der Architektur vorgebene Form haben muss, an die sich die Implementierer von Bausteinen halten müssen. Alles andere ist als *lokale Variablen* eines

jedoch genügend physische Kontrollflüsse vorhanden wären, um alle eingehenden Nachrichten eines Bausteins sofort bearbeiten zu können, wäre die Pufferung von Nachrichten überflüssig. In der Praxis ist die Bevorrung von Kontrollflüssen mit hohem Overhead verbunden; wenn man deswegen die Bevorrung von Kontrollflüssen durch dynamische Erzeugung und Destruktion von Kontrollflüssen (vgl. historisches Beispiel [Opl65]) ersetzt, landet man wieder beim allgemeinen asynchronen Modell, dessen Parallelitätsgrad theoretisch nicht beschränkt ist.

<sup>26</sup>Mittels dynamischer Code-Generierung, wie sie z.B. bei Java bei der Umsetzung von Byte-Code in Zielplattform-Maschinencode eingesetzt wird, kann man prinzipiell auch die Bindung von Prozeduraufrufen zu allen genannten Mechanismen zur Laufzeit durchführen. Es wäre ein interessantes Forschungsprojekt, dies nicht nur aus dem Quelltext oder aus Zwischencode-Darstellungen heraus, sondern auch aus bereits vollständig compiliertem Code heraus (der ggf. Zusatzinformationen enthalten müsste) zu bewerkstelligen.

Bausteins zu betrachten. Die interne Realisierung von Bausteinen darf jedoch bzw soll möglichst andere Baustein-Instanzen als lokale Variablen benutzen (wobei wiederum keine „schwarzen Schnittstellen“ eingeführt werden dürfen), so dass sich insgesamt eine baumartige *Lokalitäts-Hierarchie* ergibt, die von der hierarchischen Struktur der äußerlichen Baustein-Verdrahtungen unabhängig ist. Die Verantwortung für den Einsatz und den Betrieb von lokalen Baustein-Instanzen liegt beim Implementierer eines Bausteins. Die Tatsache der Benutzung von lokalen Baustein-Instanzen ist dabei von außen nicht sichtbar.

## 2.4 Zerlegungs- und Rekombinationsstrategien

### 2.4.1 Zerlegung der zu lösenden Aufgaben in möglichst wenige universelle Elementarteile

Bei der Problemanalyse propagiere ich die Anwendung einer einfachen Methode: es ist zu fragen, wie generische Funktionalität (vorzugsweise in Form von universeller Generizität) in möglichst wenige, aber universelle Elementarfunktionalität zerspalten werden kann. Aus dieser Elementarfunktionalität ergeben sich dann die *Elementaroperationen* als Umsetzung in ein von-Neumann-Maschinenmodell, die im Pseudo-Code einer imperativen Programmiersprache notiert werden.

### 2.4.2 Orthogonalität und Rekombinierbarkeit von Elementaroperationen

Im Idealfall sollten die Elementaroperationen die Eigenschaft der Orthogonalität besitzen, d.h. keine der Elementaroperationen sollte sich durch Kombination anderer Elementaroperationen simulieren lassen.

Eine Zerspaltung in orthogonale Elementaroperationen kann dazu führen, dass sehr kleine, nach üblichen Maßstäben „triviale“ Elementaroperationen entstehen, die in der Praxis selten oder nie als alleinstehende Operationen genutzt werden, sondern häufig nur in Kombination mit anderen Elementaroperationen. Entwerfer haben oft die konkrete programmiersprachliche Repräsentation einer Elementaroperation als Prozeduraufruf oder RPC im Hinterkopf und schrecken daher oft vor dem Overhead einer solchen „trivialen“ Zerlegung zurück. Das Problem des Overheads lässt sich jedoch weitgehend vermeiden, wenn man die *systematische Rekombination* von Elementaroperationen als eigenes Grundkonzept einführt.

Wenn die Rekombination orthogonaler Elementaroperationen bei der Bearbeiter-Instanz der Prozedur- oder RPC-Aufrufe erfolgt, ergibt sich nur ein geringer bis verschwindender Overhead im Vergleich zur direkten Implementierung jeglicher Kombination. Der Vorteil der Zerlegung besteht jedoch darin, daß im Regelfall nur der Programmcode für die wenigen Elementaroperationen implementiert werden muss, nicht dagegen für alle vorkommenden Kombinationen, die oft wesentlich zahlreicher ausfallen.

## 2.5 Das Problem der „Eierlegenden Wollmilchsau“

Größere Softwaresysteme, die über längere Zeit benutzt und erweitert wurden oder die für sehr große Anwendungsfelder und -bandbreiten ausgelegt wurden, zeigen oft ein Phänomen auf, das salopp mit „Featuritis“ bezeichnet wird. Solche Alleskönnner haben manchmal Schwierigkeiten, ganz einfache grundlegende Aufgaben auf einfache und effiziente Weise zu erledigen. Die Softwarestruktur ist im Vergleich zu einfachen Programmen, die nur die Grundaufgaben abdecken, oder im Vergleich zu Spezialisten meist deutlich aufgebläht (Overhead) und schwerer zu durchschauen.

Um dieses Problem handhabbar zu machen, schlage ich die Verwendung unterschiedlicher *Modelle* vor, zwischen denen möglichst eine hierarchische Inklusions-Beziehung in dem Sinne herrschen sollte, daß man ein Modell als Spezialfall eines anderen Modells betrachten kann; oft ist das weniger mächtige Modell auch einfacher. Als Beispiel hierfür werden in Abschnitt 3.2 verschiedene Zugriffs-Modelle unterteilt wie `singleuser` oder `multiuser` behandelt

Eine Nest- oder Baustein-Implementierung braucht nicht unbedingt alle möglichen Modelle zu unterstützen. So kann man beispielsweise mit der Implementierung der einfacheren Modelle beginnen und diese erst später und bei Bedarf auf die komplizierteren Modelle erweitern oder für die komplizierteren Modelle alternative Implementierungen vornehmen (deren höheren Kosten müssen dann von den Verwendern der einfacheren Modell-Variante nicht bezahlt werden). Welche Modelle von einer Nest- oder Baustein-Implementierung im Einzelfall unterstützt werden, wird als *Kompetenz* (*competence*) dieser Implementierungs-Instanz bezeichnet. Demgegenüber steht das *Verhalten* (*habit*) einer konkreten Aufrufer-Instanz; damit werden die Anforderungen an die Kompetenzen der Implementierungs-Instanz bezeichnet.

Die Kompetenzen einer Implementierungs-Instanz müssen das Verhalten einer Aufrufer-Instanz *kompatibel* sein. Die Kompetenzen und das Verhalten von Nest- und Baustein-Instanzen werden je Baustein-Art in Form von *Attributen* angegeben. Die Kompatibilität kann dann bei der Verdrahtung konkreter Baustein-Instanzen automatisch getestet werden, wobei inkompatible Verdrahtungen zurückgewiesen werden.

Attribute können entweder *statisch* oder *dynamisch* sein: statische Attribute eines Baustein-Typs ändern sich nie (sie hängen nur vom Baustein-Typ oder der gewählten Implementierung ab), dynamische Attribute hängen von der konkreten Instantiierung und/oder von der Verdrahtung mit anderen Bausteinen ab, ändern sich jedoch nicht während der Lebensdauer einer Baustein-Instanz. Werte, die sich während der Lebensdauer einer Baustein-Instanz ändern können, stellen keine Attribute dar, sondern gehören zum *Zustand* der Baustein-Instanz.

Attribute werden insbesondere zur Unterscheidung verschiedener Modelle eingesetzt und in Schreibmaschinenschrift gesetzt.

## 2.6 Automatismen

Die Grundidee der Automatisierung wird seit Jahrhunderten erfolgreich zur Reduktion von Aufwand und Kosten eingesetzt.

Im Kontext der hier vorgestellten Architektur bedeutet Automatisierung, dass vorzugsweise von deskriptiven Methoden Gebrauch gemacht wird, um einen Mechanismus selbsttätig auszulösen, der immer wiederkehrende Vorgänge selbsttätig bearbeitet.

Die Implementierung vieler Automatismen ist eine Frage von konkreten Strategien, die in speziellen Bausteinen (`strategy_*`) lokalisiert werden sollten (vgl. Abschnitt 4.2.2).

Zur Erstellung von Bausteinen sind weitere Automatismen von großem Nutzen, die über den Funktionsumfang üblicher Werkzeuge wie Compiler oder Debugger hinaus gehen sollten. Viele mit den hier vorgestellten (Schnittstellen-)Mechanismen zusammenhängende Konstruktionsvorgänge lassen sich automatisieren.

Die in Abschnitt 2.3.2 vorgestellte späte Bindung an konkrete Aufrufmechanismen kann beispielsweise auf folgende Weise automatisiert werden:

Der Programmierer gibt ein statisches Baustein-Attribut an, mit dem er sein Denk-Modell deklariert, mit dessen Hilfe er den Programmcode entwickelt hat. Das Attribut kann folgende Werte annehmen:

`code_nolock` Der Programmierer tut so, als gäbe es nur einen einzigen Kontrollfluss, der den Baustein betreten dürfte<sup>27</sup>; er kümmert sich also überhaupt nicht um Parallelität. Dies hat zur Folge, dass aus Sicherheitsgründen niemals ein weiterer Kontrollfluss den Baustein betreten darf, selbst wenn der bereits eingetretene Kontrollfluss eine lange dauernde blockierende Operation aufruft.

`code_monitor` Wie vorher, nur ist der Programmierer sich immerhin der Tatsache bewusst, dass mehrere Kontrollflüsse vorkommen können. Jeder Aufruf der Blockierungs-Operation `wait` (siehe Abschnitt 3.3.2) führt automatisch zu einer Entblockierung des Zugriffsschutzes gegenüber anderen Kontrollflüssen. Der Programmierer ist für die Beachtung der damit verbundenen Effekte verantwortlich; insbesondere ist ihm bewusst, dass kritische Abschnitte genau an der Stelle einer `wait`-Operation aufgehoben werden (analog zum Monitor-Konzept).

`code_reentrant` Der Programmierer ist sich dessen bewusst, dass mehrere Kontrollflüsse den Baustein asynchron betreten können. Er ist selbst für die Sicherung kritischer Abschnitte und für das Setzen von Locks verantwortlich.

Ein zu syntaktischer Analyse fähiger und die Schnittstellen-Konventionen kennender Quelltext-Präprozessor extrahiert

<sup>27</sup> Auch im `singleuser`-Modell (Abschnitt 3.2) können asynchrone Kontrollflüsse durch `notify_*`-Operationen (Kapitel 5) entstehen.

diesen Attribut-Wert aus dem Quelltext und generiert bei Bedarf (je nach eingestelltem Aufruf-Mechanismus) automatisch Lock-Operationen zur Klammerung kritischer Abschnitte, Fallunterscheidungs-Kontrollstrukturen zum Demultiplexen eingehender (L)RPC-Aufrufe, und so weiter. Auf diese Weise lässt sich jedes Programmiermodell mit jedem Schnittstellen-Mechanismus kombinieren<sup>28</sup>; bei Verwendung von statischen oder Inline-Prozeduraufrufen werden durch den Präprozessor mehrere Quelltexte verschiedener Bausteine zu einem einzigen Kombi-Baustein fest zusammengeschweisst<sup>29</sup>. Bei geeigneter sorgfältiger Konstruktion und Verwendung gut optimierender Compiler lässt sich durch Inline-Prozeduraufrufe jeglicher Schnittstellen-Aufwand fast vollständig eliminieren; dies ist insbesondere zum Anschluss von Prüf- und Sicherheits-Bausteinen oder kleineren Trivial-Bausteinen nützlich.

## 2.7 Zugriffsrechte und Schutzmechanismen

### 2.7.1 Grundlegende Betrachtung

Die Rechteverwaltung in Betriebssystemen basiert bei den meisten praktisch eingesetzten Modellen [Lan81] auf *Rechte-Relationen*, die *Subjekte* und *Objekte* miteinander in Beziehung setzen; typischerweise lassen sich derartige Relationen zwischen Subjekten und Objekten tabellarisch darstellen.

Ich möchte für die Rechteverwaltung einen Ansatz vorstellen, der sich bei geeigneter Interpretation ebenfalls als Subjekt-Objekt-Schema auffassen lässt, dabei jedoch beliebige Dinge als Subjekte bzw Objekte auffassen kann.

Zunächst ist zu fragen, was überhaupt zu schützen ist. Hierauf sind mehrere Antworten möglich. Eine möglichst allgemeine Antwort ist, dass es sich um *Informationen* handelt, die zu schützen sind. Dies umfasst die klassischen Objekte (die auf jeden Fall Informationsträger darstellen), bedeutet aber nach meiner Ansicht etwas mehr. In einem Betriebssystem gibt es Informationen, die nicht unbedingt in Form zugreifbarer Datenobjekte vorliegen müssen. Typische Beispiele hierfür sind die Menge aller möglichen Schlüsselwerte eines geheimen kryptographischen Schlüssels, oder die Rechte anderer Subjekte (die nicht unbedingt explizit repräsentiert zu sein brauchen), oder die Kompetenzen oder das Verhalten anderer Subjekte, oder andere Subjekte schlechthin; Subjekte können in anderem *Kontext* wiederum Objekte darstellen. Informationen über Objekte las-

<sup>28</sup>Falls man `code_nolock` mit (L)RPC kombiniert, braucht man selbstverständlich keine Lock-Operationen einzufügen, da es dann ja nur einen einzigen (automatisch eingefügten und bei der Initialisierung gestarteten) Kontrollfluss gibt; durch einheitliche Verwendung dieser Konfiguration bei allen Bausteinen erhält man das klassische CSP-Modell, das zu Zwecken der Fehlersuche und -Eingrenzung deutliche Vorteile bringt, ansonsten aber die potentielle Parallelität unnötig einschränkt. Die Kombination von `code_reentrant` mit (L)RPC stellt insofern eine gewisse Verschwendun dar.

<sup>29</sup>Hierbei sind (interne) Bezeichner (bzw. zu internen Bezeichnern werdende ursprünglich öffentliche Bezeichner) mit einem Versions-Kennzeichen zu verändern, um die mehrfache Instantiierung desselben Baustein-Typs in einem Kombi-Baustein möglich zu machen. Details werden in dieser Arbeit nicht behandelt.

sen sich ggf. auch durch *Schlussfolgern* oder *Ableiten* erhalten. Ich schlage deshalb vor, statt der Begriffe Subjekt und Objekt andere Begriffe zu verwenden, die durch die folgenden Überlegungen begründet sind:

Der Zweck eines Betriebssystems ist die *Ausführung von Operationen*; dies geschieht in der hier vorgestellten Architektur (vgl: Abschnitt 3.2 und Kapitel 6) durch *Bearbeiter-Instanzen* im Auftrag von *Aufrufer-Instanzen*. Da eine Aufrufer-Instanz wiederum im Auftrag mehrerer anderer Aufrufer-Instanzen handeln kann, ergeben sich zwei Fragestellungen:

1. Wer soll natürlicherweise das Subjekt darstellen, das einen bestimmten Auftrag veranlasst?
2. Wer soll natürlicherweise das Objekt darstellen, das den Auftrag ausführen soll?

Die erste Frage ist auf verschiedene Weisen beantwortbar. Daher schlage ich die Einführung eines anderen Begriffes statt „Subjekt“ vor, nämlich das *Mandat* (mandate). Operationen geschehen grundsätzlich aufgrund eines Mandates; im allgemeinen kann dabei eine Instanz (bzw ein Subjekt in bisheriger Terminologie) mehrere verschiedene Mandate *wahrnehmen*; auch kann es vorkommen, dass verschiedene Instanzen aufgrund des gleichen Mandats handeln. Beispiele hierfür finden sich in der realen Welt im Rechtswesen: ein und derselbe Rechtsanwalt kann gleichzeitig verschiedene Mandate für verschiedene Mandanten wahrnehmen. Ein Mandant kann verschiedene Mandate gleichzeitig an verschiedene Rechtsanwälte oder an den gleichen Rechtsanwalt vergeben. Er kann dasselbe Mandat aber auch an mehrere Rechtsanwälte gleichzeitig vergeben, beispielsweise bei Anwalts-Gemeinschaften oder Kanzleien. Mandate können *vertreten* oder *weitergereicht* werden. Im Rechtswesen können auch *Untermandate* vergeben und aufgeteilt werden; von der letzteren Möglichkeit habe ich hier wegen der damit verbundenen Komplexität und Folgeeffekte vorläufig Abstand genommen<sup>30</sup>. Die Menge aller möglichen Mandate sollte durch einen abstrakten Datentyp mit ausreichend grossem Wertevorrat<sup>31</sup> dargestellt werden. Mandate haben keine festliegende Interpretation, sondern werden lediglich zwischen Baustein-Instanzen weitergereicht. Alle Operationen auf Mandaten wie z.B. ihre Erzeugung gehören somit zu den Strategien, die der Implementierer eines Bausteins selbstständig bestimmen kann. Mandate stellen somit eine weitere Hilfsabstraktion dar.

Die zweite Frage ist im Kontext der hier vorgestellten Architektur relativ schnell zu beantworten: ein Objekt ist auf jeden Fall die Ausgangs-Nest-Instanz, an die der zu bearbeitende Auftrag gerichtet wird. Diese Nest-Instanz zerfällt jedoch nicht nur in Unterobjekte wie z.B. einzeln adressierbare Bytes, sondern stellt auch wegen der eingangs aufgeführten Problematik nicht alle Informationen dar, die nach

<sup>30</sup>Ob und in welchen Fällen Untermandate vorteilhafte Beschreibungen darstellen, dürfte eine interessante zu untersuchende Frage sein. Strukturell sind Untermandate zwar ähnlich zu Gruppen von Subjekten, die Erzeugung von Untermandaten kann jedoch rein dynamisch zur Laufzeit geschehen, die wiederum in weitere Untermandate zersplittet werden könnten. Ob und wie weit solche Modelle vorhersagbares und intuitiv für Menschen leicht begreifbares Verhalten zeigen, muss noch untersucht werden.

<sup>31</sup>Nach heutigen Maßstäben sind hierzu mindestens 64 Bit reservierter Platz erforderlich.

- Möglichkeit geschützt werden sollten. Die folgende Begriffsbildung kommt diesem Ziel etwas näher:

Zu schützen ist die *Ausführung* von Operationen. Die Menge aller ausführbaren Operationen wird *Operationenmenge* genannt; dies sind alle möglichen Kombinationen von Operations-Bezeichnern mit ihren Eingabe-Parameter-Werten, oder in anderen Worten die Vereinigung aller kartesischen Produkte aller Operationsnamen mit ihren Parameter-Wertemengen (egal wie viele es sein mögen). Auch wenn man die Menge aller möglichen Parameter-Versorgungen als endlich betrachtet (indem man beispielsweise die Menge aller vorkommenden Zeichenketten-Parameter beschränkt), ist die Operationenmenge i.a. sehr groß. Eine dazu äquivalente Darstellung ist die Beschreibung als Menge aller möglichen Bitstrings, die in einem generischen Operations-Nest (vgl. Kapitel 6) auftreten können.

Eine *Schutzmengen* ist eine Teilmenge der Operationenmenge eines gegebenen Systems. Schutzmengen sind als Mengen aller (im Sinne irgendeines ausserhalb definierter Zulässigkeits-Begriffes) zulässigen Operationen zu interpretieren.

## 2.7.2 Spezifikation von Schutzrechten

Der Begriff der Schutzmengen vereinfacht die Abstraktion irgendwelcher Schutz- und Zulässigkeitsbegriffe auf mathematisch sehr einfach fassbare und hochgradig flexible Weise<sup>32</sup>. Da Schutzmengen i.a. zu groß für eine tabellarische Darstellung sind, braucht man irgendwelche Spezifikationsmechanismen, die sie beschreiben. Der Zweck eines Spezifikationsmechanismus besteht einerseits darin, eine Schutzmengen für menschliche Leser verständbar und nachvollziehbar komprimiert darzustellen, und andererseits das Enthaltens-Problem der Schutzmengen speichereffizient auf Rechnern auswertbar zu machen.

Als Spezifikationsmechanismen kommen sehr viele konkrete Realisierungen (z.B. passend eingeschränkte Prädikatenlogiken oder andere Kalküle) in Betracht, deren Diskussion den Rahmen dieser Arbeit sprengen würde. Es sei nur weiteres auch militärische Schutzmodelle (vgl. [10]) oder mehrere voneinander unabhängige Schutzmodelle gleichzeitig einsetzbar.

## 2.7.3 Prüfung von Schutzrechten

Die Prüfung von Schutzrechten kann in speziellen Prüf-Bausteinen *check\_\** erfolgen, die sich prinzipiell an beliebigen Stellen einer Baustein-Hierarchie einfügen lassen. Dort weisen sie die im Sinne des jeweiligen Schutzmodells unzulässigen Operationen zurück. Falls man in speziellen Anwendungsbereichen wie z.B. Echtzeit-Steuerungen kein Schutzmodell benötigt, kann man auf die Instantiierung von *check\_\**-Bausteinen vollkommen verzichten und damit jeglichen Overhead einsparen.

<sup>32</sup>Das klassische Konzept der Rechte-Tabellen lässt sich als Spezialfall einer Schutzmengen darstellen: man bilde die Menge aller Tripel *(op,subject,object)*, die durch eine Subjekt-Objekt-Zuordnungstabelle beschrieben wird, und fülle gegebenenfalls durch die Tabelle nicht beschriebene weitere Parameter-Komponenten der Operationen mit der Menge aller möglichen Werte auf.

Die konkrete Realisierung eines Schutzmechanismus ist eine Frage der Strategie, an welchen Stellen welche Arten von Prüfungen auf Basis welcher Mandate durchgeführt werden sollen. Zu beachten ist, dass zur Laufzeit beliebige Baustein-Instanzen neu instantiiert und neue Verdrahtungen erzeugt werden können, mit denen ein festes Schutzkonzept u.U. umgangen werden kann. Beim Einsatz kompositorischer Generizität steckt ein Teil der im System vorhandenen Informationen in der Verdrahtung der Bausteine, die hochgradig flexibel und dynamisch änderbar ist. Zur Lösung dieses Problems müssen die Instantiierungs-Operationen von control-Instanzen (siehe Abschnitt 4.2.1) in das jeweilige Schutzkonzept mit einbezogen und überwacht werden. Zur mathematischen Beschreibung der dynamischen Eigenschaften von Baustein-Netzwerken eignet sich eventuell die Menge aller möglichen Baustein-Konfigurationen, die durch control erzeugbar sind.

## 2.7.4 Sicherstellung von Schutzrechten

In der Literatur über Schutzmechanismen in Betriebssystemen werden vor allem zwei Mechanismen zur Sicherstellung eines Schutzkonzepts verwendet:

1. Sicherstellung durch Zugriffsbeschränkung mittels eines Typkonzepts einer höheren Programmiersprache
2. Sicherstellung durch die MMU

Typkonzepte von Programmiersprachen haben zwar den geringsten Laufzeit-Overhead, bieten jedoch i.a. nicht das Maß an Sicherheit gegen Umgehung durch böswillige Angriffe, wie dies MMU-Hardware ermöglicht, indem sie Zugriffe auf fremde Schutzbereiche physikalisch unterbindet.

Im Prinzip lassen sich beide Mechanismen in der hier vorgestellten Architektur einsetzen, wobei Mechanismus 1 durchaus Vorteile bei der Erhöhung der Zuverlässigkeit eines Systems gegen versehentliche Fehlfunktionen mit geringem Overhead besitzt; wirkliche Sicherheit gegen bösartige Angriffe bietet jedoch nur der Mechanismus 2.

Die Realisierung eines Schutzmodells ist daher von der Einteilung des Betriebssystems in Schutzbereiche abhängig (siehe Abschnitt 4.2). Im einen Extremfall kann das gesamte System in einem einzigen Schutzbereich ablaufen, im anderen Extremfall kann jede Baustein-Instanz in einem eigenen Schutzbereich ablaufen; es sind beliebige Zwischenstufen möglich. Sinnvollerweise sollten Zugriffsrechte vor allem an den Grenzen zwischen Schutzbereichen geprüft werden, sofern dieser Aufwand in Kauf genommen werden soll.

## 2.8 Performanz-Fragen

Nicht nur in der Literatur wird der Performanz eines Systems ein hoher Stellenwert eingeräumt. Das Ziel hoher Performanz kann gelegentlich mit anderen Zielen in Widerspruch stehen. Durch konsequente Anwendung der Trennung zwischen Mechanismen und Strategien lässt sich dieser Zielkonflikt jedoch in vielen Fällen vermeiden, wenn man beim Entwurf der Mechanismen auf deren performante

Implementierbarkeit achtet (die tatsächlich erzielte Performance hängt jedoch u.a. von den Kenntnissen, Fähigkeiten und Methoden des Implementierers ab<sup>33</sup>).

### 2.8.1 Performanz von Elementaroperationen

In Abschnitt 2.4.2 wurde bereits die systematische Rekombination von Elementaroperationen aus methodischen Gründen heraus motiviert. Performanz stellt ein weiteres Motiv dar.

Aufrufer benutzen Elementaroperationen häufig in *Bündeln*, d.h. in sukzessiven Folgen, die einem bestimmten Programmier-Muster folgen. Gebündelte Aufrufe von Elementaroperationen enthalten i.A. triviale *Muster* von Programmlogik: nur bei Erfolg der ersten Operation wird die nächste aufgerufen, wobei Ergebnis-Parameter der ersten Operation an die zweite weitergegeben werden. Es kommen viele verschiedene Muster in Betracht.

Die Idee besteht nun darin, die Implementierung häufig vorkommender Muster (siehe z.B. Abschnitt 3.3.8) von der Implementierung der Elementaroperationen zu separieren. Hierfür eignen sich dedizierte Bausteine *pattern\_\**, die sich mit beliebigen anderen Bausteinen kombinieren lassen. Dies bewirkt praktisch eine Schnittstellen-Erweiterung, die jedoch keine neue Funktionalität einführt, sondern lediglich der Bequemlichkeit und teilweise auch der Performanz-Verbesserung dient. Insbesondere bei Verwendung von RPC-Mechanismen in *remote*-Bausteinen sorgt die gebündelte Übertragung einer einzigen Operation über ein Netzwerk für die Einsparung von Nachrichten und Latenzzeit, wenn sie erst beim Server in Elementaroperationen aufgespalten und interpretiert wird. Damit dies auch in größeren Baustein-Hierarchien nutzbar wird, sollten relativ einfache Bausteine wie *selector* oder *union*, die fast gar nichts tun außer Operationen mit geringen Modifikationen weiterzureichen, gebündelte Muster als passend modifizierte Bündel an die nächste Instanz weiterreichen. Dies ist jedoch keine absolute Pflicht<sup>34</sup>.

Zur Umsetzung dieser Idee sollte ein Baustein in seinen Verhaltens-Attributen die Muster angeben, die er verwenden möchte. Die Instantiierungs- und Verdrahtungs-Logik (*control* und *strategy\_\**, siehe Abschnitt 4.2) kann dann die passenden Muster-Bausteine automatisch an geeigneten Stellen einfügen und zwischenschalten, so dass die Verwendbarkeit der Muster auf jeden Fall sichergestellt ist, ohne sich darum sorgen zu müssen. Falls der untergeordnete Baustein in seinen Kompetenz-Attributen angibt, mit einem bestimmten Muster umgehen zu können, wird dieser Zwischenschritt ggf. ausgelassen. In günstigen Fällen ergibt sich dadurch eine Kette von bündelungsfähigen Verdrahtungen, die sich auch über Speicher-Hierarchien hinweg erstreckt und so die Performanz deutlich verbessert.

<sup>33</sup>In neuerer Zeit haben Performanz-Fragen ein erhöhtes Interesse in der Literatur gefunden. Trotz des Bemühens um Vergleichbarkeit verschiedener Architekturen und Vorschläge ist diese nicht immer gegeben; das Stichwort des Vergleiches zwischen Äpfeln und Birnen findet sich nicht nur in [Lie95b].

<sup>34</sup>Missachtung kann jedoch teilweise erhebliche Einbußen bei der Performanz nach sich ziehen. Wenn es dagegen auf schnelle Implementierung neuer Funktionalität ankommt, ermöglicht das Weglassen von Bündeln so genanntes „rapid prototyping“.

### 2.8.2 Zero-Copy-Architektur

Frühere IO-Schnittstellen wie die von Unix [RT74] oder [Che87] besaßen eine implizite Kopier-Semantik. Neuere Entwicklungen wie [BS96, PDZ99, PDZ00] vermeiden das Herstellen von Kopien in möglichst vielen Situationen. Es ist bekannt, dass dies dramatischen Einfluss auf die Performanz haben kann.

Die hier vorgestellte Architektur definiert daher die IO-Funktionalität der Nest-Schnittstelle mit Hilfe von Referenz-Semantik; die Übergabe von physischem Speicher über Schnittstellen hinweg erfordert dies ohnehin. Damit ist kopierfreie Übergabe von Daten über beliebig große Baustein-Hierarchien hinweg möglich. Falls in besonderen Fällen eine Kopier-Semantik gewünscht wird oder notwendig ist, kann sie leicht als interne Strategie durch einen Baustein implementiert werden.

### 2.8.3 Das Background-IO-Konzept

Durchsatz-begrenzende *Flaschenhälse* treten oft in komplexen Netzwerken von Baustein-Verdrahtungen oder in verteilten Systemen auf. Wenn ein Falschenhals auf physikalischen Bedingungen beruht (z.B. bei Festplatten-IO), lässt er sich prinzipiell nicht entfernen (außer durch Kauf schnellerer und teurerer Hardware). Man kann jedoch Methoden entwickeln, um mit derartigen Flaschenhälse besser leben zu können, d.h. zu versuchen, „das beste draus zu machen“.

IO-Prioritäten dienen diesem Zweck. Mittels IO-Prioritäten lassen sich wichtige von unwichtigen IO-Operationen trennen. Ich schlage die Verwendung folgender IO-Prioritätsstufen vor:

**prio\_background** Der IO-Auftrag kann irgendwann ausgeführt werden, beispielsweise wenn ansonsten ungenutzte Übertragungs-Bandbreite frei geworden ist. Die Wartezeit bis zur Ausführung darf beliebig lang sein; es werden keinerlei Fairness-Bedingungen garantiert oder erwartet.

**prio\_normal** Der IO-Auftrag muss prinzipiell gleichberechtigt zu anderen Aufträgen bearbeitet werden, die ebenfalls **prio\_normal** haben. Der Auftrag muss nach endlicher Zeit erledigt sein (kein Verhungern). Permutationen der Abarbeitungs-Reihenfolge sind im Rahmen dieser Bedingungen erlaubt.

**prio\_urgent** Der IO-Auftrag ist anderen vorzuziehen. Die größere Wichtigkeit hat eine logische Begründung (beispielsweise Sicherung wichtiger Meta-Daten).

Die Idee besteht darin, bei Verstopfung eines IO-Flaschenhalses Aufträge mit **prio\_background** vollkommen zu verdrängen. Bei geeigneter Realisierung lässt sich dies so gestalten, dass Hintergrund-Aufträge die normalen IO-Aktivitäten überhaupt nicht stören.

Bei Festplatten kann man dies folgendermassen realisieren: erst werden alle Aufträge mit **prio\_urgent**

und `prio_normal` abgearbeitet. Danach folgt eine spekulative Wartepause in der Größenordnung von 10ms bis 100ms, nach deren Ablauf erst mit dem Abarbeiten der Hintergrund-Operationen begonnen wird. Falls während dieser Wartepause erneute Operationen mit `prio_urgent` oder `prio_normal` eintreffen, werden *keine* Hintergrund-Operationen gestartet, sondern die höher priorisierten sofort vorgezogen; danach beginnt die Wartepause von vorn. Der Sinn dieser Wartepause besteht darin, den mechanisch bewegten Schreib-Lesekopf nicht zu verstellen, falls eine Anwendung ihre IO-Anforderungen zwar in Gruppen (Bursts) stellt, zwischen den Anforderungen aber sehr kurze Lücken (z.B. wegen kurzer Rechenzyklen) einlegt. Ohne die Wartepause könnte der mechanische Arm der Festplatte während der sehr kleinen Lücke zwischen normal-priorisierten Burst-Operationen (wenige Mikro- oder Nanosekunden) auf die Position der Hintergrund-Operation verstellt werden, was bei heutigen Festplatten im Mittel etwa 5ms bis 10ms dauert, und hernach wieder auf die Position des Burst-Prozesses zurückgestellt werden. Da das mechanische Verstellen um mehrere Größenordnungen langsamer geht als das sequentielle Lesen von Burst-Datenblöcken, würde dieses andauernde mechanische Hin-und-Herstellen zu einem Trashing-Effekt mit einem Zusammenbruch des Daten-Durchsatzes führen. Die Wartepause löst dieses Problem durch eine Spekulation auf das Eintreffen weiterer wichtiger Operationen. Wenn diese Spekulation nach einer relative langen Zeit von einigen Vielfachen der mittleren Zugriffszeit der Festplatte sich nicht erfüllt hat, dann ist es wegen der bekannten empirisch beobachteten zeitlichen Ungleichverteilung der IO-Anforderungen sehr unwahrscheinlich, dass ausgerechnet dann weitere IO-Operationen mit normaler Priorität eintreffen, während der Kopf für die Hintergrund-Aufträge verstellt wird. Falls dieser Fall trotzdem einmal eintreten sollte, dann verschlechtert er den ohnehin sehr geringen Durchsatz der normal-priorisierten Aufträge nicht mehr wesentlich.

Wenn man das Konzept eines den Normalbetrieb praktisch nicht störenden Hintergrund-IO-Betriebes in die Nest-Schleife aufnimmt, ergeben sich neuartige Realisierungen, das Problem der Persistenz über Speicherlücken hinweg. So schadet es beispielsweise in einem Buffer-Cache überhaupt nicht, wenn man jegliche geänderten Puffer *sofort* nach Bekanntwerden der Änderung (Wechsel in den „dirty“-Zustand des Puffers) einen IO-Auftrag zum Abspeichern mit Hintergrund-Priorität generiert. Falls der Hintergrund-Auftrag sehr lange nicht ausgeführt wird, dann entsteht der gleiche Effekt wie beim konventionellen Puffern ohne IO-Auftrag. Werden Hintergrund-Aufträge jedoch auf „heissen“ (d.h. oft zugegriffenen) Seiten ausgeführt, dann führt dies in der Summe zwar zu einer höheren Belastung des IO-Kanals mit Aufträgen, diese füllen jedoch nur die ansonsten ungenutzte Rest-Bandbreite des Kanals, die ansonsten „verschwendet“ worden wäre.

Im Endeffekt entsteht dadurch eine zeitnahe Sicherung transienten Speichers auf persistente Hintergrund-Medien, die sich automatisch an die *aktuell verfügbare* Bandbreite adaptiert. Bei hoher verfügbarer Bandbreite werden speulative Sicherungs-Vorgänge ausgeführt, die zu einer besseren Sicherheit gegen unvorhergesehene Ereignisse wie

Stromausfall führen.

Die Hintergrund-Priorität lässt sich weiterhin auch zum spekulativen Vor-Laden von Caches etc. nutzen. Spekulatives Ladevorgänge können wegen der Hintergrund-Priorität jederzeit von wichtigeren Vordergrund-Aktivitäten verdrängt werden. Die Realisierung solcher Preloading-Strategien kann beispielsweise durch zwischen geschaltete Verhaltens-Beobachter-Bausteine erfolgen, die das IO-Verhalten einer Anwendung beobachten und analysieren und daraus Schlüsse über das erwartete zukünftige Verhalten der Anwendung ziehen.

In Abschnitt 3.3.1 wird näher beschrieben, wie sich die IO-Priorität bereits erteilter Aufträge nachträglich erhöhen lässt. Diese dynamische Änderungsmöglichkeit ist wichtig, da richtig speulierte Anforderungen jederzeit zu „echten“ Anforderungen einer Anwendung werden können.

# Kapitel 3

## Nester

Die Abstraktion des *Nestes* stellt ein konkretes *Speichermodell* dar. Betriebssystem-Konstrukteure sind traditionell gewohnt, dass ihnen die Hardware eines Rechners oder eines Rechner-Typs ein bestimmtes Speichermodell vorschreibt oder zumindest in relativ engen Grenzen sehr nahe legt. Diesem Zwang versuche ich an einigen Stellen so zu entkommen, dass dadurch keine grundlegende Verschlechterung der Performanz-Eigenschaften der Hardware ausgelöst wird.

Eine *Nest-Instanz* ist ein *logischer Adressraum*, der von der Adresse 0 bis zu einer Maximalgröße (mit mindestens 64Bit<sup>1</sup>) laufen kann. Mit logischen Adressen darf Adress-Arithmetik getrieben werden; es werden *Bytes*<sup>2</sup> adressiert.

Ein Nest dient vor allem zur Abbildung von logischen Byte-Adressen auf *physische* Byte-Adressen von *Datenblöcken*. Damit soll u.a. das bekannte Konzept eines virtuellen Benutzer-Adressraums nachgebildet werden; dies ist jedoch prinzipiell unabhängig davon, ob MMU-Hardware in einem Rechner vorhanden ist oder nicht. Ein physischer Datenblock hat eine begrenzte Länge, innerhalb deren er sich durch Maschinenbefehle des Prozessors ansprechen lässt und innerhalb deren Adress-Arithmetik auf Basis von physischen Byte-Adressen getrieben werden darf.

### 3.1 Arten von Nestern

Es wird zwischen *statischem* und *dynamischem* Nest unterschieden: während ein statisches Nest sich ähnlich wie ein Unix-Device verhält und auch nur im Zusammenhang mit Peripheriegeräten o.ä. vorkommen sollte, basiert das gesamte restliche System auf dynamischen Nestern.

Ein dynamisches Nest kann im Unterschied zu einem konventionellen File abfragbare und veränderbare Löcher im Definitionsreich der partiellen Abbildung von logischen Adressen auf physische Adressen von Datenblöcken

<sup>1</sup>Noch größere logische Adressräume, etwa mit 128 Bit oder mehr, haben durchaus praktische Anwendungen: das bekannte Segmentierungs-Modell (vgl. Multics [Org72]) oder die Speichermodelle von Intel-Prozessoren lässt sich durch einen linearen logischen Adressraum darstellen, indem man ein (Segmentselektor,Offset)-Paar bildet, das insgesamt als logische Adresse eines (löchrigen) linearen Adressraums interpretiert wird. Wenn der Segmentselektor beispielsweise 64 Bit und der Offset ebenfalls 64 Bit umfasst, dann reicht ein logischer Adressraum von 128 Bit für eine äquivalente Darstellung des Segmentierungs-Modells durch den linearen Adressraum (vgl. auch das Thunk-Modell von Microsoft beim Übergang vom 16Bit-MSDOS-Speichermodell auf 32 Bit).

<sup>2</sup>In früheren Zeiten war der Begriff Byte mehrdeutig, da er auch andere Bitfolgen als das Oktett bezeichnen konnte. Der Begriff ist inzwischen standardisiert und bezeichnet genau 8 Bit.

enthalten, ähnlich wie ein Sparse File unter Unix (vgl. auch [Fot61, Lie95a]), jedoch wird diese Eigenschaft vom gesamten Betriebssystem auf allen Ebenen außer Low-Level-Gerätetreibern unterstützt und auch intensiv genutzt. Als neuartige Elementaroperation kommt die *transparente Verschiebung* von logischen Adressbereichen hinzu. Eine Verschiebung bewirkt, dass ein Teil des Definitionsreiches der partiellen Abbildung von logischen Adressen auf Adressen von physischen Datenblöcken so verschoben wird, dass dieselben Datenblöcke anschließend unter einem Adress-Offset (im Vergleich zu den vorigen Adressen) im logischen Adressbereich erreichbar sind. Im Allgemeinen können durch Verschiebe-Operationen Löcher im Definitionsreich entstehen und/oder geschlossen werden, eventuell können am Ziel der Verschiebung auch Datenblöcke „verloren gehen“, d.h. sie werden nicht mehr vom Nest adressiert (können aber i.a. noch solange weiter existieren, solange noch dynamische Referenzen darauf bestehen).

In konventioneller Terminologie bedeutet eine Verschiebeoperation, dass je nach Vorzeichen des Verschiebe-Offsets und Länge des verschobenen Bereichs eine Insert- oder Delete- oder Move-Operation in einer Datei, in einer Datenbank, oder allgemein gesprochen in einem Nest durchgeführt wird. Ich verwende daher nur noch die Begriffe *Nest* und *Verschiebe-Operation*, um diese Sachverhalte zu charakterisieren.

Wichtig an der Verschiebe-Operation ist, dass sie *transparent* erfolgt. Damit ist gemeint, dass nur diejenigen Teile des Gesamtsystems von einer Verschiebung Kenntnis erhalten, die unbedingt davon betroffen sind; für die anderen Teile ergeben sich durch die Verschiebung keine Änderungen.

Hierzu ein Beispiel (siehe auch Abschnitt 4.1.4): aus dem oberen Ende eines Original-Nestes sei ein Teil-Nest ausgeschnitten worden. Die Ausschneide-Operation bewirkt, dass die zugrunde liegenden Datenblöcke sowohl im Original-Nest als auch im ausgeschnittenen Teil-Nest erscheinen, dort jedoch unter neuen Adressen (im Regelfall bei 0 oder einer anderen festen Adresse beginnend). Nun finde eine Verschiebung im Original-Nest statt, die das gesamte ausgeschnittene Teil-Nest mit umfasse, der gesamte Ausschnitt also mit-verschoben werde. Transparenz bedeutet in diesem Beispiel, dass sich an den Adressen des ausgeschnittenen Teil-Nestes nichts ändert. Benutzer des Teil-Nestes merken gar nichts davon, dass ihr Gastgeber-Adressraum „hinter ihrem Rücken“ verschoben wurde.

## 3.2 Zugriffs-Modelle

Auf einer Nest-Instanz lassen sich *Operationen* ausführen, darunter *elementare Grundoperationen*. *Operations-Aufrufe* dürfen grundsätzlich parallel durch *Aufrufer-Instanzen* (wer auch immer das sein mag) bzw. *asynchron* an eine *Bearbeiter-Instanz* (i.d.R. die Implementierung der Nest-Operationen) erfolgen. Die Bearbeiter-Instanz bewirkt für alle Operations-Aufrufe jeweils zugehörige *Operations-Ausführungen*, deren ausgelösten *Effekte* die Ausführung anderer Operationen beeinflussen können. Die Beeinflussung wird durch zwei grundlegende Modelle beschrieben:

**linearize\_global** Durch Operations-Ausführung entsteht eine sequentielle Folge von *Nest-Zuständen* auf derselben Nest-Instanz. Es ist Aufgabe einer Implementierung von Nest-Operationen, für die Einhaltung dieser sogenannten *globalen Operations-Linearisierungs-Eigenschaft*<sup>3</sup> (zur Grundidee vgl. [HW90]<sup>4</sup>) zu sorgen. Verschiedene Nest-Instanzen sind grundsätzlich voneinander unabhängig, soweit nicht durch Bausteine Abhängigkeiten eingeführt werden.

**linearize\_local** Für alle Operations-Aufrufe, die sich auf eine gegenseitig überschneidende Teilmenge der logischen Adressen beziehen, wird eine sequentielle Folge von Zuständen definiert bzw. hergestellt; Operations-Ausführungen auf verschiedenen logischen Adressbereichen stehen nicht notwendigerweise in einer Ordnungsrelation zueinander (auch *Kommutierbarkeit*<sup>5</sup> genannt). Damit wird i.A. nur eine Halbordnung zwischen den Operations-Ausführungen auf derselben Nest-Instanz gat-

<sup>3</sup>Zum besseren Verständnis kann man hilfsweise den Begriff der „Folge von Operations-Aufrufen“ einführen. Dann ist die Folge der Operations-Ausführungen eine Permutation der Folge der Operations-Aufrufe, die folgende weitere Bedingung erfüllt: wenn man die beiden Folgen zu einer gemeinsamen Folge vereinigt, dann liegt jeder ursprüngliche Operations-Aufruf an seiner zugehörigen Operations-Ausführung. Dieses Modell ist nur als Hilfskonstrukt zum besseren Verständnis zu verwenden. Realiter braucht man den Begriff der Folge von Operations-Aufrufen nicht einzuführen, sondern es genügt, von Operations-Aufrufen schlechthin zu sprechen, ohne dass diese in eine Folge gebracht werden müssen. Der Grund hierfür ist nicht, weil das Herstellen dieser Folge insbesondere in verteilten Systemen gewisse bekannte Schwierigkeiten macht (vgl. [AW94]), sondern weil man es zur Beschreibung der Semantik eines rein asynchronen Modells nicht benötigt.

<sup>4</sup>Im Gegensatz zum Konsistenzmodell der Linearisierbarkeit wird hier nicht eine spezielle Speicher-Semantik unterstellt, denen die Lese- und Schreiboperationen unbedingt gehorchen müssen. Weiter unten werden beispielsweise Lese-Operationen behandelt, die ausdrücklich die Auslieferung veralteter Versionen zulassen und daher im konventionellen Modell der Linearisierbarkeit nicht enthalten sind.

<sup>5</sup>Im Allgemeinen ist die Disjunktheit der betroffenen Adressbereiche für die Kommutation von Operations-Ausführungen hinreichend, aber nicht notwendig. So genannte „semantische Modelle“ versuchen, die Auswirkungen normalerweise nicht kommutierbarer Operationen nachträglich so zu interpretieren oder zu korrigieren, dass im Sinne irgendwelcher semantischer Anforderungen oder Bedingungen ein „korrektes“ Ergebnis herauskommt. Ein derartiger Korrektheitsbegriff muss nicht notwendigerweise in allen möglichen Ausführungsfolgen den gleichen Endzustand erreichen. In Abgrenzung zum Kommutierbarkeits-Begriff möchte ich dies *Quasi-Kommutierbarkeit* nennen.

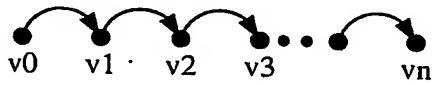
rantiert (*lokale Linearisierung*<sup>6</sup>).

Globale und lokale Linearisierung unterscheiden sich demnach nur in der Granularität, mit der die Zustände bezeichnet werden; weitere Modelle werden in Kapitel 7 behandelt. Nur wenn lokal linearisierte Operationen mit unterschiedlich großen Adressbereichen auftreten, bei denen ein größerer Adressbereich mehrere kleinere überlappt, entstehen zwangsläufig halbordnungsartige Querbezüge zwischen ansonsten unabhängigen Strängen von „Objekt“-Zuständen. Eine weitere Verfeinerung der Semantik von Nesteren mittels Zusätzen wie z.B. Bausteine zur Implementierung verschiedener Arten von Serialisierbarkeit [Vid87] ist möglich und gehört zu den Strategien, die mittels Bausteinen implementiert werden können. Das Modell der lokalen Linearisierung ist ziemlich allgemein gehalten, um verschiedene Konsistenz- oder Korrektheitsbegriffe als Spezialisierung ableiten zu können.

Die Forderung nach (irgend)einer sequentiellen Folge von Nest- oder Adressbereichs-Zuständen (unabhängig von der Granularität) lässt sich durch eine *operationalen Semantik* beschreiben. Diese legt nur fest, welche Operation welche Effekte auf einem Nest bzw. Adressbereich bewirkt, d.h. wie aus einem Zustand  $v_i$  ein Zustand  $v_j$  wird. Die Zustände  $v_i$  und  $v_j$  werden auch als *Versionen* bezeichnet. Eine Operations-Ausführung impliziert zwischen den Versionen  $v_i$  und  $v_j$  die Relation der *direkten funktionalen Abhängigkeit*. Im Allgemeinen gilt  $j > i$ , d.h. die direkte funktionale Abhängigkeit ist eine Halbordnung:



Die (oft wünschenswerte) Eigenschaft der *Determinanz*<sup>7</sup> ist dann gegeben, wenn stets  $j = i+1$  gilt, d.h. die funktionale Abhängigkeit stellt eine Totalordnung dar:



Die Folge der (globalen oder lokalen) Operations-Ausführungen hängt per Konvention mit der Folge der Nest-Zustände in einer 1:1-Beziehung zusammen (d.h. jede Operation bewirkt genau eine Version des Nest-Zustands, die ggf. gleich zu einer früheren Version ausfallen kann), auch wenn die Determinanz nicht gegeben sein sollte.

Ein Zustand  $v_i$  einer Nest-Instanz besteht aus einer *Adressabbildungs-Funktion*  $a_i$ , die logische Adressen auf

<sup>6</sup>Der Begriff *lokal* bezieht sich hierbei auf eine Eigenschaft des Nester, nämlich seinen Adressbereich, und nicht auf eine bestimmte physische Verteilung auf Rechnerknoten oder dergleichen.

<sup>7</sup>Dieser Begriff soll zur Unterscheidung von den Begriffen Determiniertheit und Determinismus dienen. Ich verwende den Begriff Nichtdeterminismus nicht, da sein Gegen teil, der Determinismus, in einem asynchronen Modell nicht auftritt (bzw. nur als Spezialfall auftritt). Determiniertheit ist auch nicht gegeben, weil i.A. die Reihenfolge der Ausführung mehrerer paralleler Operationen nicht fest liegt. Determinanz ist ein Begriff, der eine Zwischenstufe zwischen Determinismus und Nichtdeterminismus ausdrücken soll: es entsteht zwar eine „deterministische“ Folge der voneinander funktional abhängigen Zustände, doch diese Entstehung geschieht auf nichtdeterministische Weise aus Operations-Aufrufen, für die keine Ordnung voraus gesetzt wird.

physische Adressen abbildet, und einer *Datenabbildungsfunktion*  $d_i$ , die physische Adressen auf Bytes abbildet. Wir schreiben  $v_i = (a_i, d_i)$  und bringen damit zum Ausdruck, dass die Komposition der beiden Funktionen  $a_i$  und  $d_i$  den Zustand  $v_i$  vollständig charakterisiert. Die Operationen lassen sich in folgende *Operations-Arten* einteilen:

**invariant** Eine invariante Operation ändert nichts am Zustand des Nester, d.h. es gilt  $v_j = v_i$ .

**adressmodifizierend** Es gilt  $d_j = d_i$  und  $a_j \neq a_i$ . Eine adressmodifizierende Operation bewirkt, dass immer noch die gleichen Inhalte (Werte) physischer Datenblöcke in der Nest-Instanz vorhanden sind, diese jedoch teilweise unter neuen logischen Adressen erreichbar sind. Adressmodifizierende Operationen setzen ein dynamisches Nest voraus.

**datenmodifizierend** Es gilt  $a_j = a_i$  und  $d_j \neq d_i$ . Eine datenmodifizierende Operation ersetzt den Inhalt physischer Datenblöcke durch einen anderen Inhalt in einem zusammenhängenden Bereich des logischen Adressraums, der *Modifikations-Bereich* genannt wird; außerhalb des Modifikations-Bereiches ist der Effekt der Operation invariant. Datenmodifizierende Operationen können sowohl auf statische als auch auf dynamische Nester ausgeführt werden.

Was mit den Zuständen geschieht, bzw. wofür sie benutzt werden, ist eine eigenständige Frage, die sich auf verschiedene Weisen beantworten lässt. Ich propagiere folgende Modelle:

**singleuser** Es ist nur eine einzige Aufrufer-Instanz vorhanden, die Operations-Aufrufe auf der Nest-Instanz ausführen kann. Im allgemeinen kann diese Aufrufer-Instanz dennoch mehrere parallele (asynchrone) Operations-Aufrufe tätigen. Die Nest-Instanz *sichert* dem Aufrufer die Determinanz zu. Der Aufrufer hat nur Zugriff auf die jeweils *letzte* Version  $v_n$  der Zustände der Nest-Instanz, bzw. Änderungen erfolgen nur dort (Single-Copy-Eigenschaft). Bei mehreren parallelen Operations-Aufrufen durch denselben Aufrufer wird keine konkrete Reihenfolge der Operations-Ausführungen zugesichert. Falls der Aufrufer jedoch stets nur einen einzigen Operations-Aufruf in Auftrag gibt, wird ihm eine konkrete Reihenfolge zugesichert.

**multiuser** Es dürfen mehrere Aufrufer-Instanzen vorhanden sein. Diesen wird die Eigenschaft der Determinanz zugesichert. Im Vergleich zum Singleuser-Modell erhält eine einzelne Aufrufer-Instanz trotz Beauftragung mit einem einzigen Operations-Aufruf nicht mehr die Zuschreibung, in welcher Reihenfolge die Ausführung gegenüber den Aufträgen anderer Instanzen geschieht.

**multiversion** Dieses Modell wird in Kapitel 7 behandelt und zerfällt in mehrere Untermodelle. Es dürfen mehrere Aufrufer-Instanzen vorhanden sein. Diese haben im allgemeinsten Untermodell *potentiellen Zugriff* auf alle Adress-Versionen  $a_i$  und alle Daten-Versionen  $d_j$  einer Nest-Instanz, womit ein Zugriff auf beliebige *Pseudo-Zustände*  $v'_{ij} = (a_i, d_j)$  möglich ist, bei denen  $i = j$  nicht unbedingt gelten muß (was jedoch in einigen Untermodellen eventuell gefordert werden kann). Damit wird nur noch die Zusicherung gegeben, dass die Operations-Ausführung auf *irgendeiner gültigen* (also nicht auf einer undefinierten) Version der Adress- und Datenabbildung stattfindet, die jedoch auch älter sein kann.

Die Modelle **singleuser** und **multiuser** erfüllen per Konstruktion die Eigenschaft der Determinanz, haben also die Semantik von Halbleiter- oder Plattspeichern, bei denen eine Schreiboperation die vorherige Version ersetzt und nicht mehr zugänglich macht. Im Folgenden werden wir uns auf diese beiden Modelle beschränken und **multiversion** in Kapitel 7 gesondert betrachten.

Die Eigenschaft der *Korrektheit* wird durch die hier beschriebenen Nest-Modelle nicht betrachtet<sup>8</sup>; die damit verbundenen Probleme sind auf anderer Ebene zu lösen. Dies ermöglicht konkreten Baustein-Implementierungen, verschiedene Modelle von Korrektheit auf Basis der allgemeineren Abstraktion Nest zu erfüllen. Jede Aufrufer-Instanz hat daher die *Verpflichtung*, selbst<sup>9</sup> für die Erfüllung weiterer von ihr selbst gegebenen Zusicherungen, insbesondere die Korrektheit (im von ihr selbst definierten Sinne) zu sorgen. Die Abstraktion der Nester stellt jedoch dafür eine (hoffentlich bequeme) Infrastruktur zur Verfügung.

Eine Nest-Implementierung braucht nicht unbedingt alle drei Modelle zu unterstützen. Welche Modelle im Einzelfall unterstützt werden, gehört zu den Kompetenzen (vgl. Abschnitt 2.5) der Nest-Implementierung. Wie bereits gesagt, müssen die Kompetenzen der Nest-Implementierung zum Verhalten einer Aufrufer-Instanz kompatibel sein.

Die Tabelle 3.1 zeigt die möglichen Kompatibilitäten. Die Tripel bedeuten hierbei, wie viele verschiedene Aufrufer-Instanzen mit Verhalten **singleuser**, **multiuser** und **multiversion** jeweils an einer Konfiguration mit einer Nest-Implementierung der betreffenden Kompetenz teilnehmen dürfen. Während sich **singleuser** mit keinem anderen Aufrufer-Verhalten verträgt, sind **multi\*** beliebig untereinander mischbar, sofern die Nest-Implementierung von ihren Kompetenzen her mitspielen kann.

<sup>8</sup>Die einzelnen Elementaroperationen müssen natürlich trotzdem die „Korrektheit“ oder besser die „Validität“ auf ihrer jeweiligen Abstraktionsebene bzw. Zugriffs-Granularität erfüllen.

<sup>9</sup>Die Abstraktion der Nester gibt jedenfalls keine Zusicherungen über „Korrektheit“, sondern höchstens über die (Nicht-)Kommutierbarkeit von Elementaroperationen. Den Aufrufern wird überlassen, welche Semantik sie implementieren wollen; dabei dürfen und sollen sie sich jedoch auf die Zusicherungen der Nest-Implementierung abstützen.

		Verhalten		
		singleuser	multiuser	multiversion
Kompetenz	singleuser	(1, 0, 0)	-	-
	multiuser	(1, 0, 0)	(0, n, 0)	-
	multiversion	(1, 0, 0)	(0, n, m)	(0, n, m)

Tabelle 3.1: Kompatibilitäten von Kompetenzen und Verhalten

### 3.3 Elementaroperationen auf statischen und dynamischen Nestern

Der hier vorgestellte Satz von Elementaroperationen auf Nestern ist als Beispiel zu verstehen, wie man die Schnittstelle zu Nestern gestalten kann. Die Beschreibung ist informell und soll die Kernidee erklären; es handelt sich nicht um eine vollständige Spezifikation aller Details.

Der Entwurf von IO-Schnittstellen wird in der Praxis durch diverse Umstände verkompliziert, die nach meinen Erfahrungen für einen Großteil der Implementierungsqualität realer Betriebssysteme mitverantwortlich sind.

Berücksichtigung dieser Umstände bereits in den Schnittstellen und im gesamten System über alle Ebenen hinweg<sup>10</sup>, sowie durch die Unterstützung so genannter „fortgeschrittener IO-Konzepte“ wie asynchroner IO und Nonblocking-IO mittels universeller Generizität lässt sich diese Komplexität reduzieren<sup>11</sup>. Daher enthält bereits die Schnittstelle für statische Nester alle im System vorkommenden IO-Konzepte.

Ich habe versucht, im folgenden Entwurf auch die Anforderungen von Datenbanken so weit zu berücksichtigen, dass hoffentlich die Grundfunktionalität eines Datenbank-Systems abgedeckt werden kann<sup>12</sup>.

Sperrmechanismen dienen dazu, die möglichen Ausführungsfolgen von Operationen einzuschränken. Als Sperrmechanismen sind bei statischen und dynamischen Nestern die elementaren Grundoperationen lock und unlock vorgesehen, die die einzigen Sperrmechanismen sind, die im gesamten Betriebssystem benötigt werden<sup>13</sup>.

Buffer-Caches wie beispielsweise [PDZ99] vereinheitlichen zwar das Caching in einem System auf ähnliche Weise wie hier, führen jedoch ihre eigene Schnittstelle ein. Im Gegensatz dazu verlangt das Prinzip der universellen Generizität (Abschnitt 2.1.4), dass die Nest-Schnittstelle nicht verändert wird, wenn Caching irgendwo eingeführt oder nicht mehr benutzt wird.

<sup>10</sup>In der historischen Entwicklung erfolgreicher Betriebssystem-Kerne wie der UNIX-Familie wurden immer wieder so genannte „Balkone“ an die vorhandenen Konzepte angebaut, um mit geänderten und erweiterten Anforderungen Schritt halten zu können. Balkone widersprechen der Forderung nach möglichst geringer Redundanz eines Entwurfs. Die hier vorgeschlagenen Schnittstellen-Konzepte und Abstraktionen versuchen das Ideal von universeller Generizität anzunähern, um auch zukünftige unvorhersehbare Anforderungen mit etwas Glück ohne Balkone nachrüsten zu können.

<sup>11</sup>Falls fortgeschrittene Konzepte der Implementierung von Datenbank-Systemen [HRO1] nicht ausreichend berücksichtigt sein sollten, dann hoffe ich, dass sich diese Mechanismen ebenfalls in die vorgestellte Architektur mit geringem Aufwand integrieren lassen.

<sup>12</sup>Dies gilt auch für die bekannte Multiprozessor-Problematik (SMP) und die bisher durch Unterbrechungs-Sperren gelösten Synchronisationsprobleme, die durch geeignet implementierte Elementaroperationen lock und unlock lösbar sind, sofern man Unterbrechungen als vollwertige Kontrollflüsse implementiert (vgl. [KE95]).

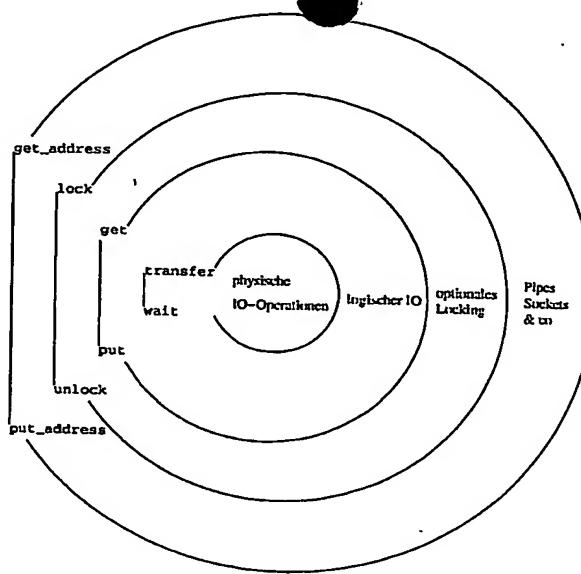
Ein statisches Nest lässt sich als Einschränkung eines dynamischen Nestes verstehen, bei dem die Verschiebe-Operation fehlt und keine Löcher im Definitionsbereich vorkommen, sowie in den meisten Fällen (etwa bei Festplatten-Partitionen) keine dynamische Änderung der Größe erfolgt. Allerdings wird diese abstrakte Sicht nicht vollkommen von der Realität aktueller Peripheriegeräte geteilt: die Granularität von Datentransfers ist bei aktueller Peripherie auf Blöcke fester Länge, etwa 512 Byte bei Festplatten oder 2048 Byte bei CD-Laufwerken, begrenzt. Darüber hinaus dürfen Datentransfers nur an solchen Adressen stattfinden, die durch die Granularität ohne Rest teilbar sind. Diese Einschränkungen werden durch ein Attribut transfer\_size modelliert, das jeder Nest-Instanz zugeordnet ist, und das jeder Verwender eines Nestes abfragen und bei seinen IO-Aufträgen berücksichtigen muss.

Bei statischen und dynamischen Nestern sorgen die Elementaroperation dafür, dass die logischen Adressen (die relativ zum jeweiligen Nest gelten) auf *physische Adressen* von transienten Datenblöcken übersetzt werden. Die Zuständigkeit für die Verwaltung der physischen Adressen kann unterschiedlich geregelt sein. Hierfür werden zwei Betriebsarten unterstützt, die prinzipiell auch gemischt nutzbar sind: *logischer IO* und *physischer IO*.

Beim logischen IO geschieht die Verwaltung (Platzreservierung und -Freigabe) der transienten Datenblöcke im physischen Hauptspeicher durch die Operation `get`, während beim physischen IO dafür der Aufrufer selbst verantwortlich ist. Physischer IO sollte idealerweise nur auf den unteren Schichten von Gerätetreibern benutzt werden, das restliche System benutzt normalerweise logischen IO. Die Unterschiede zwischen beiden Betriebsarten sind gering und werden lediglich durch eine unterschiedliche Parameter-Versorgung von `get` ausgedrückt.

Die Elementaroperationen wickeln zusammen nicht nur sämtlichen asynchronen IO ab, sondern erlauben auch den transparenten Einsatz eines Buffer-Cache ohne Änderung der Schnittstelle. Wie wir später sehen werden, führen sie auch die Speicherverwaltungs-Funktionalität des Systems durch, sogar diejenige des Paging von virtuellen Benutzer-Speicherseiten. Aus der folgenden Beschreibung des asynchronen IO-Modells ergibt sich die Realisierung von synchronem IO und Nonblocking-IO auf triviale Weise, so dass ich mir deren Ausführung sparen werde.

Das folgende Bild enthält eine Übersicht über die Elementaroperationen auf statischen Nestern:



Wie zu sehen ist, werden die Elementaroperationen immer paarweise mit einer entgegengesetzten oder aufhebenden Operation kombiniert. Verschiedene Arten von IO oder sonstige Zugriffs-Arten auf Nester ergeben sich durch ein Schichtenmodell, das hier kurz von innen nach außen skizziert werden soll:

Im einfachsten Fall wird physischer IO durch die Operation `transfer` betrieben. Die Funktionalität synchronen IOs entsteht durch anschließendes Warten auf die Beendigung mittels `wait`. Bei asynchronem IO kann auch auf den Aufruf von `wait` verzichtet werden. Bei logischem IO wird die Abbildung von logischen auf physische Adressen von der Operation `get` übernommen. Durch den paarweisen Aufruf von `put` wird die Ressourcen-Verwaltung auf ähnliche Weise wie bei einem konventionellen Buffer-Cache gelöst (interne Verwendung von Referenzzählern). Damit ein Ausgang eines Bausteins mit mehreren Eingängen anderer Bausteine parallel verdrahtet werden darf (multiuser-Kompetenz), wird der wechselseitige Ausschluss (vgl. [Lag78]) bei parallelem Zugriff durch die Operationen `lock` und `unlock` bereit gestellt. Zur Nachbildung von Pipes u.ä. sowie zur Speicherverwaltung existieren weitere Elementaroperationen `get_address` und `put_address`, mit denen das Problem der atomaren Reservierung von Adressbereichen insbesondere im Falle mehrerer paralleler Zugreifer lösbar ist.

### 3.3.1 transfer

Diese Elementaroperation sorgt dafür, dass eine *Aktualisierungs-Operation* in eine der beiden möglichen Richtungen zwischen logischer Adresse im Nest und physischem Datenblock beauftragt wird (ohne auf die Ausführung dieser Operation zu warten). Eine derartige Aktualisierungs-Operation wird synonym *IO-Operation* oder *IO-Auftrag* genannt; anstelle des Begriffs *IO-Operations-Ausführung* wird synonym auch der Begriff *Beendigung* des *IO-Auftrags* verwendet. Verschiedene *IO-Aufträge* sind prinzipiell voneinander unabhängig. Die Schnittstelle sieht folgendermaßen aus:

```
transfer(nest, mandate, log_addr, len,
phys_addr, direction, io_prio,
```

`depend) → success`

Der Parameter `direction` kann einen der folgenden Werte annehmen:

<code>read</code>	Die Aktualisierung erfolgt von der logischen Adresse im Nest in Richtung zum physischen Datenblock. Nach erfolgreicher Beendigung des IO-Auftrages wird zugesichert, dass sich die neueste Version an der physischen Adresse befindet.
<code>write</code>	Die Aktualisierung erfolgt in Richtung von der physischen Datenblock-Adresse an die logische Adresse auf dem Hintergrund-Medium. Mit Durchführung des IO-Auftrages entsteht eine geänderte neueste Version des Nest-Zustandes.
<code>stop</code>	Ein eventuell bereits wartender IO-Auftrag wird wieder aus der Warteschlange der <i>wartenden</i> Aufträge entfernt, sofern dies möglich ist; bei bereits in Ausführung befindlichen IO-Aufträgen ist dies im Regelfall nicht mehr möglich. Die Folge ist, dass die von den anderen <code>direction</code> -Arten gegebenen Zusicherungen nicht mehr eingehalten werden müssen. Falls ein <code>read</code> -Auftrag abgebrochen wurde, steht anschließend <code>max_version</code> auf <code>undefined</code> .

Der IO-Auftrag erfolgt mit der Prioritätsangabe `io_prio`, die in Abschnitt 2.8.3 ausführlicher erklärt wird. Im Regelfall werden Aufträge mit einer höheren IO-Priorität vor solchen mit geringerer Priorität ausgeführt; Ausnahmen sind möglich, wenn dadurch der Gesamtdurchsatz des IO-Systems gesteigert wird. Der Parameter `depend` erlaubt die Spezifikation einer Halbordnung zwischen IO-Aufträgen, was insbesondere zur Implementierung absturzsicherer atomarer Operationen bzw. Transaktionen benötigt wird. Es wird eine Liste derjenigen logischen Adressen angegeben, deren zugehörige IO-Aufträge (sofern sie existieren) auf jeden Fall beendet sein müssen, bevor der aktuelle IO-Auftrag ausgeführt (physikalisch gestartet) werden darf.

Falls `transfer` auf einen bereits vorhandenen IO-Auftrag trifft, sollte ein doppelter Eintrag von gleichlauenden Aufträgen vermieden werden; statt dessen wird lediglich ein intern zugeordneter Referenzzähler erhöht (dieser muss erst ab multiuser-Kompetenz vorhanden sein). Dadurch wird eine Überflutung des IO-Systems durch unnötige IO-Aufträge selbst bei massenhaftem Einsatz von Background-IO (vgl. Abschnitt 2.8.3) verhindert. Ggf. wird die IO-Priorität eines bereits wartenden Auftrages auf das neue Niveau erhöht, so dass in Zukunft mit einer zügigeren Bearbeitung zu rechnen sein wird.

Im boolschen Ergebnis-Parameter `success` wird zurückgemeldet, ob ein erfolgreiches Erzeugen des IO-Auftrages möglich war.

### 3.3.2 wait

Diese Elementaroperation implementiert mittelfristiges Warten. Die Schnittstelle sieht folgendermaßen aus:

```
wait(nest, mandate, log_addr, len,
      pause, keep) →
      (min_version, max_version)
```

Falls der boolesche Parameter `pause` gesetzt wurde und (noch) ein IO-Auftrag existiert, der noch nicht ausgeführt (physikalisch beendet) wurde, wird auf die Beendigung gewartet. Der boolesche Parameter `keep` legt fest, ob der interne Referenzzähler eines vorhandenen Auftrages (selbst nach erfolgreicher Beendigung) belassen werden oder dekrementiert werden soll; beim Belassen ist eine Statusabfrage durch nachfolgende `wait`-Operationen derselben Aufrufer-Instanz weiterhin möglich. Durch entsprechende Kombinationen der `pause`- und `keep`-Parameter ist Polling möglich. Aufträge, deren Referenzzähler zu 0 geworden ist, dürfen bzw sollen entfernt werden.

Ergebnis-Parameter `min_version` und `max_version` melden die Aktualität nach der folgenden Systematik:

`undefined` Der Inhalt des Datenblocks an der zurückgelieferten physischen Adresse ist undefined.

`newest` Die Nest-Implementierung sichert zu, dass die neueste Version des Nest-Zustandes im Datenblock ausgeliefert wurde, die zum Zeitpunkt der Operations-Ausführung gültig war. Sofern keine Sperrmechanismen die Ausführung weiterer Operationen verhindern, wird nicht garantiert, dass in der Zwischenzeit zwischen der Operations-Ausführung und der Inspektion durch die Aufrufer-Instanz keine weiteren Operations-Ausführungen stattgefunden haben, die den Nest-Zustand inzwischen verändert haben.

Zwischen diesen Werten gilt die Relation `undefined < newest`. Weiterhin gilt `min_version ≤ max_version`. Der Wert von `min_version` bedeutet, dass sich die Nest-Implementierung dessen sicher ist, dass der Datenblock mindestens die angegebene Aktualität besitzt. Umgekehrt bedeutet `max_version`, dass die Nest-Implementierung sicher ist, dass die Version den angegebenen Aktualitätsgrad nicht überschreitet. Wenn die Nest-Implementierung beispielsweise überhaupt nichts über die Aktualität weiß, dann ist `min_version = undefined` und `max_version = newest`. Nur wenn `min_version = max_version = newest` gilt, ist der Datenblock aktuell. Aus `min_version = max_version = undefined` kann gefolgert werden, dass entweder kein IO-Auftrag (mehr) vorhanden ist, oder dass sein Effekt durch `transfer` im stop-Modus wieder aufgehoben wurde, oder dass die `transfer`-Operation technisch nicht erfolgreich war (Schreib- oder Lesefehler).

### 3.3.3 get

Ein Buffer-Cache verwaltet eine *transiente interne Zuordnung* zwischen logischen und physischen Adressen. Diese Zuordnung ist *transient*, d.h. sie gilt längstens für den Zeitraum, in dem das Betriebssystem läuft, und kann wieder aufgelöst werden, sobald keine dynamischen Referenzen mehr auf den physischen Datenblock bestehen. Trotzdem sollte die Zuordnung nach Möglichkeit längere Zeit bestehen, als der Aufrufer unbedingt verlangt. Ohne die Möglichkeit zur Aufbewahrung einer zeitlich länger dauernden Zuordnung zwischen logischen und physischen Adressen wäre eine *Caching-Funktionalität* nicht oder nur schlecht erfüllbar.

Die Elementaroperation `get` verwaltet daher eine *transiente interne Zuordnung* zwischen logischen und physischen Adressen. Die Schnittstelle umfasst folgende Parameter:

```
get(nest, mandate, log_addr, len, mode) →
      (phys_addr, len, min_version, max_version)
```

Die übergebenen Adressen müssen zur erfolgreichen Durchführung durch `transfer_size` teilbar sein, die Länge muss ebenfalls ein Vielfaches davon darstellen. Als Ergebnis wird die physische Adresse zurückgeliefert, sofern die Operation erfolgreich war, ansonsten eine NULL-Adresse. Die zurückgelieferte Länge darf kürzer als die angeforderte Länge sein (beispielsweise wenn das unterliegende IO-System keine Transfers mit der gewünschten Länge am Stück ausführen kann), sie muss aber durch `transfer_size` teilbar sein.

Die Nest-Instanz sucht auf atomare Weise in ihren internen Zuordnungs-Strukturen (z.B. schnelle Hash-Tabellen), ob bereits eine Zuordnung mit einem physischen Datenblock an der gewünschten logischen Adresse besteht. Falls ja, dann wird diese Adresse und die ggf. gekürzte Länge zurückgeliefert. Falls noch keine Zuordnung besteht, wird Platz für einen neuen zusammenhängenden Datenblock der Länge `len` allokiert, oder falls dies nicht möglich ist, ein möglichst großer Datenblock bis zur Länge `len` (in Vielfachen von `transfer_size`). Schließlich werden die Adresse und Länge in die Zuordnungs-Strukturen eingetragen und zurückgeliefert. Die Ergebnis-Parameter `min_version` und `max_version` sind wie bei `wait` definiert.

Physische Adressen von Datenblöcken stellen *Aliase* dar, die ab multiuser-Kompetenz von mehreren Aufrufern gemeinsam benutzt werden können. Es wird allerdings nicht garantiert<sup>14</sup>, dass verschiedene Aufrufer-Instanzen beim `get` an der gleichen logischen Adresse dieselbe physische Adresse zurück erhalten<sup>15</sup>. Die Benutzung physischer Adressen geschieht durch Maschinenbefehle, die von den Operationen der Nest-Implementierung unabhängig sind und wegen der Aliase prinzipiell zu einer *Interferenz* führen können, wenn nicht bestimmte Regeln eingehalten werden. Das Reglement sieht folgendermaßen aus:

Jedem zurückgelieferten physischen Datenblock ist ab multiuser-Kompetenz ein *Referenzzähler* zugeordnet,

<sup>14</sup> Umgekehrt wird jedoch ebenfalls nicht garantiert, dass *kein* Alias entsteht.

<sup>15</sup> Dies ist in Verteilten Systemen im allgemeinen schlicht unmöglich.

der die Anzahl der auf ihn existierenden Referenzen verwaltet. Ein Aufruf von `get`, der einen bereits vorhandenen Datenblock auffindet, inkrementiert diesen Referenzzähler. Falls ein neuer Datenblock allokiert wurde, dann steht der Referenzzähler regelmäßig auf 1. Die logische Freigabe des physischen Datenblocks erfolgt durch die später beschriebene Operation `put`, die den Referenzzähler wieder dekrementiert und daher immer paarweise zu `get` aufgerufen werden muss (gleiche Anzahl von Aufrufen).

Der `mandate`-Parameter wird erst ab dem `multiversion`-Modell benötigt (siehe Kapitel 7).

Der boolsche Parameter `mode` gibt an, ob der Aufrufer den Datenblock anschließend zu Schreibzwecken benutzen darf. Bei einigen Implementierungs-Paradigmen (vgl. Abschnitt 2.7) wird diese Information genutzt, um Verstöße gegen das Reglement zu erkennen und ggf. zu sanktionieren. Ab `multiuser`-Kompetenz wird durch einen weiteren internen Referenzzähler gemerkt, wieviele Aufrufer den `mode`-Parameter gesetzt hatten.

Falls mehrere Aufrufer den `mode`-Parameter gesetzt haben, dürfen diese prinzipiell unabhängig voneinander eigene Änderungsoperationen auf dem physischen Datenblock ausführen, was zu Interferenzen führen kann<sup>16</sup>, wenn sie keine weiteren Maßnahmen ergreifen (z.B. durch Setzen von Locks<sup>17</sup>).

### 3.3.4 put

Diese Operation dekrementiert den internen Referenzzähler, der jedem physischen Datenblock zugeordnet wird, auf atomare Weise. Die Schnittstelle lautet:

```
put(nest, mandate, phys_addr, mode)
```

Falls der Referenzzähler durch diese Operation zu 0 wird, dann darf die ursprünglich durch `get` hergestellte Zuordnung zwischen logischen und physischen Adressen wieder auf atomare Weise aufgelöst werden und der Speicherplatz für den Datenblock freigegeben werden. Dies soll jedoch möglichst nur bei Speichermangel geschehen. Nach dem letzten `put` haben Zugriffe auf die physische Adresse keine Bedeutung mehr für die Aufrufer-Instanz.

Falls irgendeine Instanz Zugriffe auf eine physische Adresse macht, die keine Bedeutung (mehr) für sie hat, dann gilt dies als *Fehler*. Falls Fehler nicht abgefangen und

<sup>16</sup>Interferenzen brauchen nicht generell zu entstehen; beispielsweise könnten externe Absprachen zwischen mehreren Schreibern bestehen, dass diese nur auf zueinander disjunkte Bereiche eines physischen Datenblocks ändern zugreifen; ein typischer Anwendungsfall dafür ist z.B. ein Nachrichtenpuffer zur wechselseitigen Publikation von irgendwelchen Status-Informationen, die keinen Einfluss auf die Korrektheit haben, sondern z.B. zur Lastbalancierung genutzt werden. Die hier vorgestellte Architektur möchte die Lösung von Interferenz-Problemen durch Baustein-Implementierungen ermöglichen und unterstützen, aber nicht erzwingen, insbesondere denjenigen Implementierern keine Hemmschuhe in den Weg legen, die „wissen was sie tun“ und die nicht für Performanz-Overhead durch höherwertige Zusicherungen bezahlen möchten. Im übrigen lassen sich höherwertige Zusicherungen jederzeit durch geeignete Bausteine nachrüsten.

<sup>17</sup>Interferenzen sind auch zwischen Nur-Lesern und einem Schreiber möglich. Die Nest-Implementierung übernimmt dafür keine Verantwortung. In Abschnitt 3.3.5 wird beschrieben, wie Baustein-Implementierungen durch Locks für die Vermeidung von Interferenzen sorgen können.

verhindert werden könnten (vgl. Abschnitt 2.7), dann kann sich als Folge daraus das gesamte System in einem undefinierten Zustand befinden. Jegliche Instanzen haben daher die Pflicht, Fehler zu vermeiden.

Durch den Parameter `mode` wird der interne Referenzzähler für die Anzahl der Schreiber verwaltet; Aufrufer haben die Pflicht, diesen Parameter paarweise übereinstimmend zum `mode`-Parameter von `get` zu verwenden (Zu widerhandlungen gelten als Fehler).

### 3.3.5 lock und unlock

Sperren bewirken, dass die möglichen Ausführungs-Folgen von Operationen eingeschränkt werden. Die in unserem Modell verwendeten Sperren wirken nicht nur gegenüber anderen Sperren, sondern auch gegenüber einigen<sup>18</sup> anderen Elementaroperationen. Implementierungen von Bausteinen müssen für die Realisierung irgendwelcher Korrektheits-Begriffe grundsätzlich selbst Sorge tragen, indem sie die möglichen Ausführungsfolgen einer Nest-Instanz mit den zulässigen Ausführungsfolgen ihres jeweiligen Korrektheits-Begriffes *verträglich* machen. Dies belastet zwar die Baustein-Implementierungen mit der Pflicht, für diese Verträglichkeit selbst zu sorgen, ermöglicht dafür jedoch wesentlich flexiblere Korrektheits-Modelle. Ist eine Sperre erst einmal erfolgreich gesetzt worden, dann garantiert die jeweils beauftragte Ausführungs-Instanz (Nest-Implementierung) ihre Einhaltung gegenüber der Aufrufer-Instanz.

Die hier vorgeschlagenen Sperren sollen neben der Grundfunktionalität eines Betriebssystems auch wichtige Bereiche der Funktionalität von Datenbanken und Transaktionen (siehe z.B. [Dat95, EN95]) abdecken. Das hier vorgestellte Modell ist als *Vorschlag* zu verstehen, der an manchen Stellen verbessert werden kann (beispielsweise durch Einführung intentionaler Locking-Arten).

Die Elementaroperationen `lock` und `unlock` sperren bzw entsperren *Bereiche des logischen Adressraums* gegenüber anderen Aufrufer-Instanzen der `lock`-Operation bzw. gegenüber anderen Mandats-Inhabern und gegenüber `get`; gehaltene Sperren gelten grundsätzlich als der aufrufenden Instanz bzw den in Abschnitt 2.7 beschriebenen Mandaten zugeordnet. Die gesperrten Bereiche müssen nicht notwendigerweise im Definitionsbereich von dynamischen Nestern aktuell vorkommen („Phantom-Locks“); es findet also keine Synchronisation auf Daten-Objekten statt, sondern eine auf *logischen Adressbereichen*. `lock`-Operationen, die auf gegenseitig nicht überlappenden Adressbereichen ausgeführt werden, sind voneinander unabhängig und kommunizieren miteinander. Locks sind nicht rekursiv, sondern *akkumulierend* innerhalb derselben Lock-Art. Es schadet nicht, wenn bereits erhaltene Lock-Bereiche von derselben Instanz bzw unter dem gleichen Mandat nochmals angefordert werden.

<sup>18</sup>Bei Betriebssystem-Schnittstellen wird zwischen den Locking-Arten *mandatory* und *advisory* unterschieden. Die hier verwendeten Locks gehören zu keiner dieser beiden Arten. Es wird zwar ähnlich wie beim *advisory locking* niemand gezwungen, Locks zu verwenden; falls jedoch eine Aufrufer-Instanz einen Lock erhält, dann wird seine Einhaltung auch gesichert, und zwar auch dann, wenn sich andere Instanzen „nicht an die Regeln halten“. Im Gegensatz zum *advisory locking* wird eine Umgehung des Locks durch nicht-konformes Verhalten anderer Instanzen verhindert.

dert werden, bewirkt aber auch nichts. Die Rückgabe mittels `unlock` braucht nicht den gleichen Adressbereich zu betreffen als der vorherige `lock`. Damit ist es möglich, einen ursprünglich am Stück liegenden gesperrten Bereich in kleinere Teilstücke mit zwischenliegenden freigegebenen Bereichen umzuwandeln. Umgekehrt darf ohne Schaden ein größerer Bereich mittels `unlock` freigegeben werden als vorher mittels `lock` gesperrt wurde; falls man z.B. `unlock` mit dem gesamten Adressraum als Parameter aufruft, dann werden sämtliche von der aufrufenden Instanz bzw dem Mandat gehaltenen Locks atomar auf einen Schlag freigegeben (*starker Zweiphasen-Commit*). Die Parameter lauten:

```
lock(nest, mandate, log_address, len,
      try_address, try_len, kind, action) →
      (locked_address, locked_len)
unlock(nest, mandate, log_address, len)
```

Der *Sperr-Adressbereich* wird durch die Parameter `log_address` und `len` vorgegeben. Die Parameter `try_address` und `try_len` beschreiben einen *Versuchs-Adressbereich*, der einen Oberbereich des Sperrbereichs darstellen muss. Falls der Versuchsbereich echt größer als der Sperrbereich ist, dann braucht er von der Nest-Implementierung nur so weit im *Ergebnis-Sperrbereich* `locked_address` und `locked_len` berücksichtigt zu werden, wie dies ohne *zusätzliches* Warten möglich ist. Nähere Erklärungen für das damit mögliche *spekulative Locking* finden sich in Abschnitt 5.3. Der Ergebnis-Sperrbereich umfasst entweder mindestens den Sperr-Adressbereich (d.h. er liegt zwischen dem Sperr- und dem Versuchs-Bereich), oder er hat die Länge 0. Im letzteren Fall war ein Setzen des Locks nicht möglich (s.u.). Durch den Parameter `kind` werden neben den bekannten Arten *Read-Lock* und *Write-Lock*<sup>19</sup> noch weiter verfeinerte Lock-Arten unterschieden. Er hat einen der folgenden Werte:

**read** Der Ergebnis-Sperrbereich ist anschließend vor *Überschreiben* der  $d_i$ -Komponente durch andere Aufrufer-Instanzen von `lock` und von `get` jeweils im *write-Modus* *geschützt*<sup>20</sup>.

**write** Der Ergebnis-Sperrbereich ist anschließend zum *Überschreiben* der  $d_i$ -Komponente für den Aufrufer *exklusiv reserviert*.

**upgrade** Wie `write`, jedoch wird ein bereits gesetzter Read-Lock zu einem Write-Lock *atomar aufgewertet* (sofern es möglich ist); dieser Modus wird bei einigen Transaktions-Modellen (insbesondere pessimistische Modelle) benötigt und sollte ausserhalb dieser Anwendung nur mit Vorsicht eingesetzt werden<sup>21</sup>.

<sup>19</sup>Vgl. Semantik von File-Locking in neueren Unix-Abkömmlingen [Vah96], die sowohl die Semantik von Mutex-Semaphoren als auch von read/write-Locks aus der Datenbank-Welt als Spezialfall umfasst

<sup>20</sup>Im Fachgebiet der Datenbanken wird diese Eigenschaft als *repeatable read* bezeichnet.

<sup>21</sup>Falls mehrere Instanzen jeweils einen erfolgreichen Read-Lock auf derselben Version halten, dann kann nur eine einzige von ihnen diesen erfolgreich in einen Write-Lock umwandeln, da ansonsten der Schutz gegen Modifikation durch andere Instanzen (*repeatable read*) verloren geht.

fix

Der Ergebnis-Sperrbereich ist anschliessend gegen Änderungen der  $a_i$ -Komponente durch andere Aufrufer-Instanzen geschützt.

reorg

Der Ergebnis-Sperrbereich ist anschliessend zum Ändern der  $a_i$ -Komponente für den Aufrufer exklusiv reserviert.

Die Arten `fix` und `reorg` entsprechen in ihrer Semantik den konventionellen Arten `read` und `write`, sie beziehen sich jedoch auf Modifikationen der Adressbereiche (z.B. durch `move`).

Der Parameter `action` beschreibt, wie die `lock`-Operation mit dem gewünschten Lock umgehen soll. Er hat einen der folgenden Werte:

ask

Es wird lediglich nachgesehen, ob das Setzen eines entsprechenden Locks ohne zu blockieren möglich gewesen wäre. Der zurückgelieferte Ergebnis-Sperrbereich kann an Wettrennen mit anderen Aufrufern von Lock teilnehmen und ist entsprechend vorsichtig zu behandeln.

try

Der Lock wird atomar gesetzt, wenn es sofort ohne zu blockieren möglich ist. Ansonsten wird die Länge 0 zurückgeliefert.

wait

Der Lock wird gesetzt, auch wenn dazu erst auf die Freigabe durch eine andere Instanz gewartet werden muss. Die Rücklieferung von `locked_len = 0` ist trotzdem möglich, insbesondere wenn ein Deadlock vorliegt. Das Ergebnis muss daher in jedem Fall vom Aufrufer überprüft werden.

Die Einschränkung der möglichen Ausführungsfolgen hängt von den Kompetenzen der Nest-Implementierung ab. Bei `singleuser` brauchen keine Locks implementiert zu werden; `multiversion` wird in Kapitel 7 behandelt. Bei `multiuser` sieht die Kompatibilitätstabelle der Operationen folgendermaßen aus:

	gr	gw	lr	lw	lu
gr	+	+	+	-	*
gw	+	+	-	*	-
lr	+	-	+	-	*
lw	-	*	-	-	-
lu	*	-	*	-	-

hen müsste. Bei klassischen Transaktionen stört dies nicht grundlegend, da die anderen Transaktionen, die dies ebenfalls versuchen, einfach mittels Rollback abgebrochen werden. Ausserhalb von Transaktionen besteht jedoch das Problem, dass ein Update auf die neueste Version nicht durch Transaktions-Abbruch erfolgen kann, sondern „von Hand“ erledigt werden muss (i.d.R. durch Freigabe des inzwischen veralteten Read-Locks, erneutem Lock und `transfer` im `read`-Modus). Wer diesen Modus einsetzt, muss also damit rechnen, dass er nicht immer funktioniert, weil es genau genommen eine *Spekulation* darstellt, bei der darauf spekuliert wird, dass niemand ausser der eigenen Instanz eine Modifikation der gesperrten Daten vornehmen wird. Diese Spekulation lässt sich sicherlich auch ausserhalb von Transaktionen vorteilhaft zur Erhöhung der potentiellen Parallelität einsetzen, dies wird jedoch durch erhöhte Komplexität der Programm-Logik erkauft.

Der erste Buchstabe der Überschriften-Bezeichnung bezeichnet die Operation `g=get` oder `l=lock`. Der zweite Buchstabe bezeichnet die Lock-Art `r=read`, `w=write` oder `u=upgrade`. Im Falle von `get` bezieht sich der zweite Buchstabe auf den mode-Parameter. Es sind die logischen Adress-Zuordnungen sämtlicher gehaltenen Datenblöcke in Betracht zu ziehen, die sich mit dem angeforderten lock-Bereich überschneiden.

In der Tabelle bedeutet `+`, dass die Operation ohne zu warten durchläuft, wenn sie auf die jeweils andere stößt, und `-` bedeutet, dass gewartet werden muss, bis die andere Operation aufgehoben wurde (d.h. es findet eine Einschränkung der möglichen Ausführungsfolge statt). Das Zeichen `*` bedeutet, dass es davon abhängt, ob die andere Operation von der gleichen Aufrufer-Instanz initiiert wurde; falls ja, dann wird nicht gewartet (andernfalls würde ja sofort ein Deadlock entstehen); falls nein, dann muss gewartet werden. An denjenigen Stellen der Tabelle, wo ein `-` steht, darf die andere Operation, wenn sie von der aktuellen verschieden ist, nicht von der selben Aufrufer-Instanz ausgeführt werden sein, sondern nur von einer anderen<sup>22</sup>.

Die Lock-Arten `fix` und `reorg` sollen gegen die Effekte der später beschriebenen adressmodifizierenden Operationen `move`, `clear` und `delete` schützen, daher sind diese in der folgenden Tabelle berücksichtigt:

	<code>m</code>	<code>lf</code>	<code>lg</code>
<code>m</code>	<code>+</code>	<code>-</code>	<code>*</code>
<code>lf</code>	<code>-</code>	<code>+</code>	<code>-</code>
<code>lg</code>	<code>*</code>	<code>-</code>	<code>-</code>

In der Überschrift bedeutet `m=move` oder `clear` oder `delete` (irgendeine adressmodifizierende Operation). Der Endbuchstabe `f` bedeutet `fix`, `g` bedeutet `reorg`. Wie man sieht, funktionieren die Adressänderungs-Sperren analog zu den klassischen Read-Write-Locks, nur beziehen sie sich ausschliesslich auf die  $a_i$ -Komponenten der Zustände.

Die Realisierung von `lock` und `unlock` ist intern so zu gestalten, dass auf eine Trennung in Spinlocks und Kontrollfluss wechselnde Locks verzichtet wird (vgl. [K+91, LA93]), und fernerhin eine Benutzung durch Unterbrechungen (interrupts) möglich ist (vgl. [KE95]).

### 3.3.6 Freiwillige Selbstkontrolle des Verhaltens

Eine Nest-Implementierung garantiert durch die hier propagierten Lock-Mechanismen wettern-freies Verhalten und Schutz gegen Eingriffe anderer Instanzen, und zwar auch dann, wenn andere sich nicht an Locking-Konventionen halten. Diese Garantie nützt jedoch einer Aufrufer-Instanz mit `multi*`-Verhalten nur dann etwas, wenn sie die Locks auch tatsächlich benutzt – wer das „vergisst“, der ist selbst an möglichen Folgen schuld. Bei einem komplexeren System sind jedoch Fehler, die von fehlenden oder falsch gesetzten Locks verursacht werden, extrem schwer reproduzierbar und zu finden. Jede Baustein-Implementierung soll-

<sup>22</sup>Andernfalls stellt es einen (ggf. erkennbaren und per Fehlercode sanktionierbaren) Fehler dar.

te daher ihr Verhalten durch eine freiwillige Selbstkontrolle mittels folgender Tabelle prüfen:

	<code>go</code>	<code>gr</code>	<code>gw</code>	<code>lo</code>	<code>lr</code>	<code>lw</code>	<code>lu</code>	<code>lf</code>	<code>lg</code>
<code>go</code>				<code>-</code>	<code>-</code>	<code>-</code>			
<code>gr</code>				<code>-</code>	<code>-</code>	<code>-</code>			
<code>gw</code>				<code>-</code>	<code>-</code>	<code>-</code>	<code>-</code>		
<code>lo</code>	<code>!</code>	<code>-</code>	<code>-</code>	<code>-</code>	<code>-</code>	<code>-</code>	<code>-</code>		
<code>lr</code>	<code>-</code>	<code>!</code>	<code>-</code>	<code>-</code>	<code>-</code>	<code>-</code>	<code>-</code>	<code>!</code>	
<code>lw</code>	<code>-</code>	<code>!</code>	<code>!</code>	<code>-</code>	<code>-</code>	<code>-</code>			
<code>lu</code>	<code>-</code>	<code>!</code>	<code>!</code>	<code>-</code>	<code>-</code>	<code>-</code>			
<code>lf</code>	<code>&amp;</code>	<code>-</code>							
<code>lg</code>	<code>&amp;</code>	<code>-</code>							

Hierbei bedeutet `g=get` und `l=lock`. Der letzte Buchstabe hat die gleiche Bedeutung wie bei den Kompatibilitäts-Tabellen. Die Zeilen bezeichnen die erste Operation, die Spalten die zweite Operation. Beide Operationen beziehen sich auf dieselbe Instanz, die sie aufrufen. Die Tabelle ist daher nicht symmetrisch wie die vorherigen Tabellen, sondern zeigt die Wichtigkeit der Reihenfolge der Operationen. In der Tabelle bedeutet `!`, dass die zweite Operation nur dann aufgerufen werden darf, wenn die erste bereits ausgeführt wurde (und noch nicht durch eine Gegenoperation aufgehoben wurde). Wenn in einer Spalte mehrere `!` auftauchen, dann reicht es, wenn eine der ersten Operationen vorher ausgeführt wurde. Das Zeichen `&` bedeutet, dass die betreffende erste Operation zusätzlich ausgeführt werden muss, wobei eine der mit `&` markierten Operationen ausreichend ist. Das Zeichen `-` bedeutet, dass die erste Operation auf gar keinen Fall vorher aufgerufen werden darf, bzw. dass sie inzwischen wieder aufgehoben worden sein muss.

Obwohl diese Verhaltens-Selbstkontrolle freiwillig ist, wird sie de facto als Standard für das Verhalten aller Bausteine betrachtet; Ausnahmen sind z.B. zur Inspektion irgendwelcher Zustände oder zu Statistik-Zwecken zulässig, sollten aber begründet werden (z.B. damit, dass der betreffende Vorgang sowieso inhärent wettern-freies ist). Die Selbstkontrolle kann z.B. durch interne Verwendung eines Prüf-Bausteins erfolgen, der alle Operations-Aufrufe mitliest und bei Verletzung der Tabelle Warnungen oder Fehlermeldungen ausgibt, ggf. auch zu härteren Sanktionen greift. Da die Information über gehaltene Locks auf jeden Fall in der Bearbeiter-Instanz der Locks geführt werden muss, bietet sich diese ebenfalls als Ort zur performanten Durchführung der an sie delegierten Kontrolle an. Falls die Aufgabenstellung extreme Anforderungen an die Performance stellt, kann die Selbstkontrolle zu Lasten erhöhter Fehleranfälligkeit auch ausgeschaltet werden.

### 3.3.7 `get maxlen` und `set maxlen`

`get maxlen(nest) → len`

`set maxlen(nest, len) → success`

Damit lässt sich die maximale Länge (maximale benutzbare Adresse plus 1) eines Nestes abfragen bzw. setzen. Bei einigen (eher selten vorkommenden) Geräten lässt sich diese Länge verändern (z.B. bei Bandlaufwerken durch Schreiben

einer EOF-Marke oder dergleichen); bei den meisten Geräten liegt die Länge jedoch physikalisch fest.

Größere Bedeutung hat `set_maxlen` bei dynamischen Nestern, wobei sowohl Verkleinerungen als auch Vergrößerungen die Semantik von `delete` in den betroffenen Bereichen ergeben (siehe Abschnitt 3.4.3).

### 3.3.8 Stream-IO, Pipes, Sockets und Speicherverwaltung

Die bisher beschriebenen Elementaroperationen lösen die Probleme des portionsweisen Zugriffs, des asynchronen IO über Speicherhierarchien hinweg, und der damit verbundenen Speicherverwaltung. Dies funktioniert jedoch nur dann, wenn die logischen Adressen *bekannt sind*, an denen IO geschehen soll. Bei rein sequentiell IO, wie er z.B. bei sequentiellen Streams, Pipes oder Sockets auftritt, benutzen konventionelle Schnittstellen keine Adressen in der Schnittstelle.

Nester lassen sich zur Nachbildung von Pipes und Sockets als Ringpuffer<sup>23</sup> (mit durch `get_maxlen` beschränkter Länge) auszuführen. Logische Adressen werden der Schnittstelle verwendet, die jedoch nicht vom Aufrufer, sondern von der Implementierungs-Instanz der Nest-Operationen vergeben und verwaltet werden. Damit werden weitere Nutzungsarten und Kombinationen möglich<sup>24</sup>, insbesondere der *Lookahead* auf noch nicht endgültig konsumierte Daten.

Das Problem der *Atomarität der Reservierung* ist eine besondere Eigenschaft von Pipes, Sockets und regulären Dateien im Append-Modus. Parallele Write-Operationen stören sich nicht gegenseitig und überschreiben sich nicht, sondern führen letztlich stets zu *irgendeiner* serialisierten Folge von Schreiboperationen (auch wenn die konkrete Reihenfolge i.a. nicht determiniert ist). Ebenso ist beim parallelen Lesen aus Pipes und Sockets garantiert, dass ein und dieselben Daten bei irgendeinem der Leser genau einmal ankommen werden (auch wenn der konkrete Empfänger wiederum nicht determiniert ist). Auf Nester übertragen, bedeutet dies, dass *keine Wettrennen* zwischen verschiede-

<sup>23</sup> So genannter „Wraparound“ des zirkulären Puffers lässt sich auch durch Einschieben einer move-Operation vermeiden, sobald der Platz am Ende des Puffers nicht mehr reicht.

<sup>24</sup> Getrennte Adressbekanntgabe ermöglicht u.a. eine einfache Lösung für ein Problem, das insbesondere bei Unix zu einer Aufblähung der Systemschnittstellen geführt hat: wenn ein Konsument unter Unix wissen möchte, ob Daten an einem IO-Kanal anliegen, kann er dies mit Hilfe der Systemaufrufe `select` bzw `poll` erfahren; i.a. kann er jedoch nicht die Größe der Daten erfahren; eine Ausnahme ist lediglich die Socket-Schnittstelle, die Lookahead auf Datenpakete unter gewissen Umständen ermöglicht. Diese historisch später nachgerüstete Funktionalität zeigt ein offenbar vorhandenes Bedürfnis nach dieser Funktionalität auf.

Man kann unter Unix nicht ohne weiteres Daten „auf Verdacht“ oder mehrmals inspizieren (Lookahead), um dann aufgrund ihres Inhaltes zu entscheiden, ob man sie wirklich „konsumieren“ oder zur Verarbeitung durch eine andere Instanz im Puffer belassen will. Dass eine solche Funktionalität auch auf regulären Dateien und Pipes gebraucht wird, ist daran zu sehen, dass sie in der Standard-Benutzerbibliothek `libc` teilweise angeboten wird. Wenn diese Funktionalität nicht erst dort simuliert würde, sondern bereits im Kern integriert wäre, dann würden gewisse durch die doppelte Pufferung im Kern und Benutzer-Adressraum bedingte Anomalien nicht auftreten. Dazu gehört beispielsweise, dass ein Lookahead nicht zwischen verschiedenen Prozessen funktioniert, die aus der gleichen Pipe lesen wollen.

nen Aufrufer-Instanzen (z.B. Prozessen) bei der *Inspektion von Änderungen* in der Addressabbildung stattfinden dürfen. Wenn man andererseits den Lookahead auf noch nicht endgültig konsumierte Daten ermöglichen will, dann muss man solche Wettrennen wiederum ermöglichen (es ist dann Aufgabe der Konsumenten, mit den damit verbundenen Effekten korrekt umzugehen).

Zur Lösung der Atomarität des Reservierungs-Problems schlage ich vor, die Operationen `get_address` und `put_address` einzuführen. Die Schnittstelle sieht folgendermaßen aus:

```
get_address (nest, min_len, max_len,
            where, lock, action) → (log_address, len)
```

Die beiden Längen-Parameter kennzeichnen das Minimum und das Maximum eines Reservierungs-Wunsches. Der boolsche Parameter `where` gibt an, ob die Reservierung im bisher undefinierten Bereich der Adressabbildung stattfinden soll (Write-Funktionalität) oder im definierten Bereich (Read-Funktionalität). Falls `action = wait` ist und eine Reservierung der Minimallänge nicht möglich ist, blockiert der Aufruf so lange, bis zumindest die Minimal-Reservierung möglich ist (langfristiges Warten). Der boolsche Parameter `lock` bestimmt, ob mehrere Aufrufer parallel jeweils einen reservierten Adressbereich erhalten dürfen, oder ob weitere Bewerber bis zum nachfolgenden `put_address` warten müssen.

Als Ergebnis wird die Adresse des nunmehr reservierten Bereiches zurückgeliefert sowie seine tatsächliche Länge, die zwischen der Minimal- und Maximal-Anforderung liegen kann (in Vielfachen der `transfer_size`); bei Fehlschlagen wird 0 zurückgeliefert. Die Nest-Implementierung sichert jeder Aufrufer-Instanz die Atomarität der Reservierung zu, indem parallele Aufrufe von `get_address` in jedem Falle disjunkte Adressbereiche ausliefern. Der Aufrufer besitzt anschließend Kenntnis über einen nur für ihn exklusiv reservierten Adressbereich, mit deren Hilfe er Datenblöcke lesen oder schreiben, und/oder Änderungen an der Addressabbildung mittels `clear` oder `delete` (siehe Abschnitte 3.4.2 ff.) ausführen kann<sup>25</sup>.

```
put_address (nest, len)
```

Hebt die Reservierung wieder auf. Jeder Aufrufer von `get_address` hat die Pflicht, einen einmal erhaltenen Adressbereich irgendwann wieder mittels `put_address` freizugeben; der Zeitpunkt und die Reihenfolge bleibt dabei prinzipiell offen.

Die Standard-Pipe-Funktionalität wird durch folgenden Ablauf erzielt:

Leser: der Aufrufer führt als erstes `get_address` aus, dann `get` im erhaltenen Adressbereich; falls die Aktualität der Daten noch nicht gegeben sein sollte, folgt `transfer` im read-Modus und `wait`. Nach der Verarbeitung der Daten folgen `delete`, `put` und `put_address`.

<sup>25</sup> Die atomare Reservierung führt dies jedenfalls nicht selbsttätig aus. Es ist Sache des Aufrufers, wofür er den reservierten Bereich benutzt.

Schreiber: der Aufrufer führt als erstes `get_address`, dann `clear` und `get` aus. Nach der Ablage der Daten folgen `transfer` im `write`-Modus, `put` und `put_address`.

Falls ein Schreiber oder Leser die Daten nicht endgültig konsumieren oder bereitstellen will, kann er auf `delete` bzw `clear` auch verzichten, bzw. die jeweilige Adressraum-Operation durch eine Kompensations-Operation vor dem `put_address` wieder aufheben. Ob und wie viele Daten tatsächlich konsumiert oder bereitgestellt wurden, wird ausschließlich durch Änderungs-Operationen des Definitionsbereiches angezeigt, die von der Reservierung auch nach unten abweichen dürfen. Der Standard-Fall einer vollen Ausnutzung sollte sich durch das Konzept der systematischen Rekombination von Elementaroperationen (vgl. Abschnitt 2.4) effizient lösen lassen.

Wenn der Parameter `lock` nicht gesetzt ist, dann kann die tatsächliche Belegung des Adressraums durch `clear` bzw die Freigabe durch `delete` auch in einer anderen Reihenfolge stattfinden, als die Reihenfolge der Adress-Reservierung vorgab. Es kann sogar passieren, dass gar kein lückenlos zusammenhängender definierter Adressbereich entsteht, wenn einige der Aufrufer nach `get_address` auf die entsprechende Manipulation des Adressraums verzichten oder sie wieder rückgängig machen. Dadurch geht u.U. die Reihenfolge-Beziehung der Datenpakete verloren, die bei der Pipe-Funktionalität notwendig ist, anderen Anwendungen aber die Möglichkeit einer *out-of-order*-Behandlung bietet<sup>26</sup>.

Wenn man die `transfer`-Operationen weglässt, erhält man eine nichtpersistente Semantik. Damit kann man die Funktionalität eines Speicher-*Heaps* nachbilden. Daher eignet sich diese Schnittstelle auch für jegliche Speicherverwaltung, sogar für interne Objekte fester Größe<sup>27</sup>.

## 3.4 Elementaroperationen auf dynamischen Nestern

Dynamische Nester enthalten alle Operationen eines statischen Nestes sowie zusätzlich die folgenden adressmodifizierenden Operationen. Sie haben mit der Lückenhaftigkeit des Definitionsbereich der Speicherabbildung und mit der Verschiebeoperation zu tun.

Falls versucht wird, auf Löchern im Definitionsbereich eines Nestes zu lesen oder zu schreiben, werden Fehlercodes zurückgeliefert. Dies unterscheidet Nester von der Semantik von Sparse Files unter Unix. Falls letztere Semantik gewünscht wird, ist diese relativ einfach durch einen entsprechenden Anpassungsbaustein implementierbar.

### 3.4.1 `get_map`

Diese Operation liefert Informationen über die definierten Bereiche eines dynamischen Nestes. Konzeptionell ist dies

<sup>26</sup>Beispielsweise zur Verwaltung von IO-Aufträgen für Festplatten-Geräte treiber, wo Umordnungen der Reihenfolge zur Durchsatzsteigerung eingesetzt werden.

<sup>27</sup>so genannte Objekt-Caches lassen sich sehr einfach dadurch realisieren, dass man die Objekt-Größe als `transfer_size` benutzt.

eine Liste von Paaren, wobei jedes Paar die logische Startadresse eines definierten Bereichs und seine Länge repräsentiert. Repräsentiert wird diese Liste durch ein Array von Paaren, das nach den Startadressen aufsteigend sortiert gehalten wird. Zur effizienten Suche nach Adressen kann daher beispielsweise binäre Suche verwendet werden.

Da ein solches Array in Extremfällen sehr lang werden kann, wird es von `get_map` nicht direkt zurück geliefert, sondern statt dessen wird ein Verweis auf eine weitere Nest-Instanz zurück geliefert, die dieses Array enthält, und von der nur gelesen werden darf.

`get_map(nest) → nest`

Die von `get_map` zurückgelieferte Nest-Instanz wird *adjungiertes Nest* genannt. Sie enthält den Definitionsbereich des Original-Nestes in der oben beschriebenen Repräsentation. Adjungierte Nester haben ihrerseits auf jeden Fall keine Löcher, d.h. ihr adjungiertes Nest hat nur noch eine triviale Struktur mit einem einzigen Eintrag<sup>28</sup>

Diese Lösung hat den Vorteil, dass die interne Verwaltung des Definitionsbereiches durch lokale Baustein-Instanzen geschehen kann, die das adjungierte Nest direkt verwalten. Erweiterungen und Löschungen im Definitionsbereich lassen sich in diesem Fall direkt durch move-Operationen auf dem adjungierten Nest realisieren. Ansonsten dürfen Implementierungen von Nestern ihren Definitionsbereich auch auf beliebige andere Weise intern realisieren, sofern sie ein virtuelles adjungiertes Nest zur Verfügung stellen, das seinen Inhalt auf Anforderung dynamisch generiert.

Die im adjungierten Nest enthaltene Liste hat eine besondere Bedeutung im Falle von Geräten mit dynamischen Block- oder Sektorgrößen (insbesondere Bandlaufwerke, bei denen jeder Datenblock eine individuelle Größe haben kann, oder sonstige zeichenorientierte Geräte, bei denen es auf die Länge von einzelnen Transfers ankommt), bei Pipes und bei Netzwerk-Sockets, bei denen die Paketgröße eines einzelnen Original-Eintrags vom Empfänger ermittelbar sein muss<sup>29</sup>. In all diesen Fällen repräsentiert das adjungierte Nest dicht aneinander liegende Adressbereiche (ohne dazwischenliegende Lücken). Damit kann jeder Verwender selbst entscheiden, ob er einen Unix-typischen Zugriff auf das Nest unter Missachtung der ursprünglichen Paket- bzw. Datensatzgrenzen machen will, oder ob er zuvor die Paket- bzw. Datensatzgrenzen im adjungierten Nest nachsehen und entsprechend berücksichtigen will.

<sup>28</sup>Mit `get_map(s)` sei das zu einem beliebigen Nest *s* gehörige adjungierte Nest notiert. Die *n*-fache Komposition der `get_map`-Funktion mit sich selbst sei mit `get_mapn` notiert. Dann gilt für *n* > 2: `get_mapn(s)` = `get_mapn+1(s)`. Das dreifach adjungierte Nest beschreibt sich also selbst.

<sup>29</sup>Das von BSD stammende Socket-Konzept und seine Schnittstelle ist heute in praktisch allen bedeutenden Betriebssystemen realisiert, auch in den Betriebssystemen von Microsoft, in allen modernen Unix-Varianten, und ist selbst in neueren Versionen von IBM-Großrechner-Betriebssystemen nachgerüstet worden. Da Sockets je nach Modus und Umständen das Lesen von Paketen sowohl mit Beachtung der beim Schreiben vorgegebenen Paketgrenzen, als auch nach dem Unix-Paradigma mit aufgelösten Paketgrenzen als unabhängige Folge von Teilblöcken unterstützen und verlangen, kommen ernsthafte Betriebssystem-Entwürfe im Endeffekt nicht darum herum, beide Paradigmen zu unterstützen. Wenn man schon beide Paradigmen unterstützen muss, dann sollte dies auf einheitliche Weise, bei möglichst vielen Arten von Nestern und auf möglichst allen Ebenen geschehen.

- Ein dichtes Aneinanderliegen von definierten Bereichen ist auch bei persistenten Nestern möglich. Die Benutzer werden jedoch gebeten, derlei in der Praxis möglichst zu meiden, da es zu Verständigungsproblemen mit den Datei-Paradigmen anderer aktueller Betriebssysteme kommen könnte<sup>30</sup>.

Konzeptuell gesehen ist ein Nest somit weit mehr als ein File unter Unix, da es untrennbar mit seinem Definitionsbereich verknüpft ist, der bei der hier vorgestellten Architektur in einem adjungierten Nest repräsentiert wird.

Bei der Repräsentation eines Unix- oder Windows-Files in einem Nest enthält das zugehörige adjungierte Nest nur einen einzigen Eintrag mit der Startadresse 0 und der Länge des Files<sup>31</sup>. Die mittels `get_maxlen` und `set_maxlen` zugreifbare Maximaladresse hat hierbei nur die Funktion einer Sicherheitsschranke ähnlich einer Quota, die nichts über den tatsächlichen Platzbedarf aussagt und ohne weiteres auch Werte wie z.B.  $2^{63} - 1$  annehmen kann.

#### 3.4.2 clear

`(nest, log_address, len, mode) → success`

Damit wird der Definitionsbereich eines Nestes an der angegebenen Stelle und Länge erweitert, soweit sich vorher dort eine Lücke befand. Falls anschließend in diesem Bereich des Adressraumes gelesen wird, erscheinen Null-Byte-Datenblöcke.

Dies gilt auch dann, wenn vorher bereits Datenblöcke dem betroffenen Bereich ganz oder teilweise zugeordnet waren. In diesem Fall bleiben bereits mittels `get` ausgelieferte Datenblöcke weiterhin verwendbar, allerdings werden später abgesetzte oder bereits gepufferte `transfer`-Operationen auf diesen Blöcken nicht mehr tatsächlich ausgeführt. Nach endgültiger Freigabe durch `put` werden derartige *verwaiste (orphaned)* Datenblöcke sofort zum Speicher-Recycling verwendet.

Da ein `get` auch Blöcke mit Vielfachen der `transfer_size` ausliefern kann, die von einem nachfolgenden `put` eventuell nur zu einem Teil betroffen sind, muss die interne Logik der Nest-Implementierung sicherstellen, dass bei nachfolgenden Operationen (insbesondere `put` und `transfer`) der nicht betroffene Teil eines solchen Blocks wie ein normaler Block, der betroffene Teil dagegen als verwaist behandelt wird.

Falls `clear` einen bereits definierten Bereich berührt oder mit ihm ganz oder teilweise überlappt, dann findet standardmäßig eine Erweiterung bzw. Verschmelzung der Definitionsbereiche im adjungierten Nest statt. Ggf. werden Löcher auch vollständig aufgefüllt und benachbarte Bereiche miteinander zu einem größeren Bereich verschmolzen.

Durch Setzen des boolschen Parameters `mode` lässt sich dieses Standard-Verhalten so abändern, dass genau an den Grenzen des `clear`-Bereiches Einträge im adjungierten Nest so erzeugt bzw. verändert werden, dass der Bereich als ein einziges Datenpaket mit ggf. lückenlosem Anschluss

<sup>30</sup> Dieser Mechanismus kann auch dazu dienen, File-Paradigmen aus anderen (meist älteren) Betriebssystemen nachzubilden, die nicht das Unix-Paradigma vom File als Folge von Bytes verfolgen.

<sup>31</sup> In diesem Fall gilt bereits für  $n > 1$ :  $\text{get\_map}^n(s) = \text{get\_map}^{n+1}(s)$ .

an (ggf. erst dadurch entstandene) benachbarte Bereiche erscheint. Falls vorher im `clear`-Bereich mehrere kleinere definierte Bereiche gelegen waren, dann werden sie durch diese `clear`-Variante ebenfalls „platt gemacht“.

#### 3.4.3 delete

`delete(nest, log_address, len) → success`  
Diese Operation bewirkt, dass an der angegebenen Stelle und Länge ein Loch im Definitionsbereich entsteht bzw. bereits vorhandene Löcher ggf. erweitert werden. Falls betroffene Datenblöcke bereits vorher mittels `get` ganz oder teilweise in Verwendung genommen worden waren, dann werden diese bei den IO-Operationen und beim Speicher-Recycling analog zu `clear` als verwaist behandelt.

#### 3.4.4 move

`move(nest, log_address, len, offset) → success`

Ein durch die angegebene Adresse und Länge bezeichneter Bereich des Adressraums wird um den Offset (positiv oder negativ) verschoben. Der Inhalt der verschobenen Datenblöcke wird hierbei nicht verändert, lediglich die Zuordnung von logischen zu physischen Adressen wird verändert. Bereits mittels `get` ausgelieferte Blöcke, die vollständig im Quellbereich der Verschiebung liegen, bleiben samt ihren physischen Adressen weiterhin gültig, jedoch wird ihre interne transiente Zuordnung gemäß der Verschiebung angepasst. Beim späteren `put` werden sie so behandelt, als hätten sie schon immer an der neuen Adresse gelegen. Quell- und Zielbereich der Verschiebung dürfen sich unabhängig von der Verschieberichtung überlappen<sup>32</sup>; gegenüber `fix`- und `reorg`-Locks sind grundsätzlich beide Bereiche als relevant zu betrachten, auch wenn sie sich nicht überlappen. Im freigewordenen Quellbereich erscheint ein Loch. Falls sich im Zielbereich bereits definierte Bereiche befinden, werden diese „verdeckt“; bereits durch `get` ausgelieferte Datenblöcke werden analog zu `delete` als verwaist behandelt. Eventuelle im Quellbereich vorhandene Löcher werden mitverschoben; die Freigabe von bereits vorhandenen Datenblöcken im Zielgebiet erfolgt selbst dann, wenn ausschließlich ein Loch dorthin verschoben wurde<sup>33</sup>.

Falls vor dem `move` bereits Blöcke mittels `get` angefordert worden waren, die vollständig im Quellbereich lagen, dann werden sie anschließend so behandelt, als hätten sie schon bei der Anforderung im Zielbereich gelegen. Bei Blöcken mit einer vielfachen Länge der `transfer_size`, die von der Verschiebung nur teilweise betroffen sind, muss die interne Verwaltungslogik die interne

<sup>32</sup> Eine Variante von `move` könnte den überdeckten Bereich des Zielgebietes in den gleichgroßen freigewordenen Bereich des Quellgebietes in einer einzigen atomaren Operation verschieben, so dass im Endeffekt eine Art „Rotationsoperation“ ohne jeglichen Verlust von Daten ausgeführt wird. Damit lässt sich insbesondere eine Vertauschung von Adressbereichen realisieren. Im Moment sehe ich für diese Variante nur geringe praktische Notwendigkeit, jedoch bildet diese Variante von `move` zusammen mit `clear` und `delete` eine Algebra von Operationen auf Adressräumen mit sehr schönen mathematischen Eigenschaften.

<sup>33</sup> Würde man `co` als zulässige Quell- und Ziel-Angaben für die Verschiebung einführen, könnte man damit nebenbei auch die Semantik von `clear` und `delete` erfüllen.

transiente Zuordnung aufspalten und getrennt behandeln, so dass die einzelnen Teile bei folgenden Operationen wie z.B. `put` oder `transfer` so behandelt werden, als wären sie ursprünglich nicht am Stück angefordert worden.

Eine effiziente Realisierung von `move` ist möglich und wird in Abschnitt 4.1.3 (Baustein `map_simple`) näher detailliert.

### 3.4.5 `get_meta`

`get_meta(nest) → nest`

Zur Verwaltung von Meta-Information über den Zustand einer Nest-Instanz kann dieser eine Hilfs-Nest-Instanz zugeordnet sein, die *Meta-Nest* genannt wird. Falls das Meta-Nest nicht existiert, wird ein NULL-Kennzeichen zurückgeliefert.

Meta-Nester sind dazu gedacht, um Informationen über ein Nest bereitzustellen, analog zu Datei-Attributen oder Inode-Informationen (siehe [Bac86]), jedoch nicht ausschließlich auf diese Verwendungszwecke beschränkt. Meta-Nester werden später in Abschnitt 6.1 eine herausragende Rolle bei der Einführung von Typsystemen spielen, ebenso in Abschnitt 4.2 bei der Auto-Instantiierung von Bausteinen und in Abschnitt 4.1.5 bei den Lokalitätseigenschaften von Zugriffen auf Verzeichnis-Hierarchien.

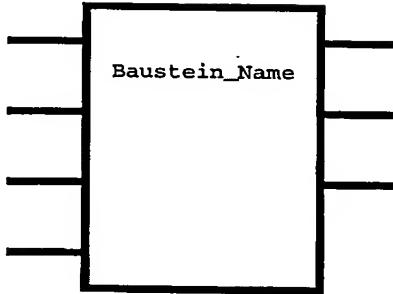
Ein Meta-Nest enthält nur einen einzigen definierten Bereich ohne Löcher, der idealerweise nur relativ wenig Platz beanspruchen sollte (d.h. er sollte nicht zur Speicherung großer Datenmengen missbraucht werden), um ein performantes Zugriffsverhalten sicherzustellen. Wegen der beabsichtigten Kleinheit sollte ein Meta-Nest eine `transfer_size` von 1 unterstützen.

Bei einem persistent gehaltenen Original-Nest muss das zugehörige Meta-Nest, sofern es existiert, ebenfalls persistent gehalten werden. Falls absturzsichere atomare Transaktionen implementiert werden, muß das Meta-Nest bezüglich dieser Semantik wie ein Teil des Original-Nestes behandelt werden.

Meta-Nester lassen sich grundsätzlich weiterhin zur Repräsentation statischer und dynamischer Attribute benutzen; dies führt zu einer Vereinfachung der zu implementierenden Konzepte, macht die Anwesenheit (virtueller) Meta-Nester jedoch zur Pflicht.

# Kapitel 4

## Bausteine



Eine Baustein-Instanz ist ein Objekt, das eine beliebige, eventuell auch zur Laufzeit sich ändernde Anzahl von Ein- und Ausgängen besitzt, die wiederum jeweils Instanzen von Nestern darstellen.

Bausteine werden ähnlich wie in der Elektro- und Digitaltechnik als Kästchen mit linksseitigen Eingängen und rechtsseitigen Ausgängen gezeichnet. Als Verdrahtungsregel gilt, dass Eingänge nur mit den Ausgängen anderer Bausteine verknüpft werden dürfen und umgekehrt, wobei ein Eingang nur mit einem einzigen Ausgang, ein Ausgang hingegen i.A. mit mehreren Eingängen verknüpft werden darf.

Ausgänge stellen damit Nester anderer Bausteinen zur Verfügung, wobei deren Eingänge als *Konsumenten* der von den Ausgängen angebotenen *Dienstleistungen* anzusehen sind; wir haben also ein System von *Produzenten* und von *Konsumenten* im *logischen* Sinne. Eine Verdrahtungs-Leitung präsentiert eine *Hierarchie-Beziehung*<sup>1</sup> zwischen einem *nachgeschalteten* Baustein (Produzent der Dienstleistung) und einem *nachgeschalteten* (Konsument der Dienstleistung). Per Konvention definieren wir als „Stromrichtung“ einer Verdrahtungs-Leitung die Richtung vom Produzenten zum Konsumenten; dies hat jedoch nichts mit möglichen Datenfluss-Richtungen zu tun, denn die Stromrichtung ist zwar gleichzeitig die logische Datenfluss-Richtung beim Lesen, doch die logische Schreib-Datenflussrichtung läuft dem entgegen (was am Anfang zu Verwirrung führen kann, ebenso die baustein-interne Weitergabe von Operations-Aufrufen, die intern meistens von den Ausgängen her zu den Eingängen verläuft). Eine Leitung stellt alle Operationen, die am Ausgang eines Bausteins zur Verfügung gestellt werden, einem oder mehreren Konsumenten zur Verfügung.

<sup>1</sup>Dies hat hohe Ähnlichkeit mit einem Schichtenmodell. Ein reinrassiges Schichtenmodell lässt sich z.B. durch eine Kette von Baustein-Instanzen nachbilden. Im Unterschied zu Schichtenmodellen werden auch neben-läufige Hierarchien und graphenartige Strukturen unterstützt, die von einer Baumstruktur abweichen. Im Allgemeinen sind jedoch keine zyklischen Verdrahtungen erlaubt.

Da dies auch die Operationen `get_map` und `get_meta` umfasst, wird auf diese Weise das zugehörige adjungierte Nest und das Meta-Nest verfügbar gemacht.

Sinn der Bausteine ist, verschiedene Transformationen sowohl des Adressbereiches als auch eventuell des Inhalts von Nestern, gelegentlich auch des zugehörigen Meta-Nestes oder von Nest-Attributen wie `transfer_size` oder von Zugriffs-Modellen durchzuführen. Durch Kombination der Bausteine zu komplexen Netzwerken ergeben sich Transformationsmöglichkeiten, die mit herkömmlichen Betriebssystem-Architekturen nur sehr schwer realisierbar sind.

Bausteine besitzen mindestens eine Instantiierungs- und Konstruktor-Operation, mit der sich neue Instanzen des jeweiligen Baustein-Typs erzeugen lassen, wobei die Parameter von Konstruktor-Operationen i.d.R. baustein-spezifisch sind. Die Destruktor-Operation hat hingegen eine einheitliche Schnittstelle. Einige Baustein-Typen haben darüber hinaus weitere spezifische Operationen, mit denen sich ihr Verhalten (etwa die Anzahl der Ein- und Ausgänge) zur Laufzeit steuern lässt.

Eine automatisierte Überprüfung der Kompatibilität der Kompetenzen von Ausgängen mit dem Verhalten von Eingängen findet bei der Verdrahtungs-Operation statt; beispielsweise wird das Zusammenpassen des `transfer_size`-Attributs überprüft. Weitere Details dazu in Abschnitt 4.2. Viele der nachfolgend vorgestellten Bausteine lassen sich in verschiedenen Ausbaustufen<sup>2</sup> und mit verschiedenen Kompetenzen und Verhalten implementieren.

### 4.1 Beispiel-Baustein-Arten

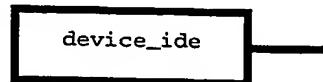
Beim Entwurf von Bausteinen sollte darauf geachtet werden, dass möglichst wenig Redundanz zur Funktionalität bereits vorhandener Baustein-Arten auftritt. Sinn der Zerlegung in Bausteine ist, die in einem Betriebssystem insgesamt zu lösenden Aufgaben in möglichst viele, kleine, von einander möglichst unabhängige (orthogonale) Teile und Zuständigkeiten aufzuspalten.

Ich habe mich bemüht, bei der hier als beispielhaft zu verstehenden Zerlegung möglichst nur solche Aufgaben zu stellen, die auch in anderen aktuellen Betriebssystem-Entwürfen (einschließlich Netzwerk-Betriebssysteme) oder

<sup>2</sup>Wenn ein neu entwickelter Baustein im singleuser-Modus stabil läuft, dann kann er schrittweise bis zur multiversion-Kompetenz bzw. -Verhalten erweitert werden.

in Datenbanken irgendwo auftauchen und dort mit teils hohem Aufwand ad hoc gelöst werden. Damit möchte ich zeigen, dass eine Komponenten-Zerlegung auf Basis der Abstraktionen Nest und Baustein den Gesamtaufwand der zu implementierenden Aufgaben mindert, da bereits die Kombination von wenigen wiederverwendbaren Baustein-Arten eine mächtige Funktionalität erzeugt.

#### 4.1.1 device\_\*



Diese Bausteine haben im Regelfall keinen Eingang<sup>3</sup> und genau einen Ausgang, womit sie den Inhalt eines Gerätes (beispielsweise device\_ide für IDE-Festplatten) als statisches Nest den etwaigen Konsumenten zur Verfügung stellen. Im Sinne der Verdrahtungslogik der Bausteine stellen sie Produzenten oder auch „Datenquellen“ dar.

Eine prinzipielle Unterscheidung zwischen Block- und Character-Devices wie bei Unix ist nicht notwendig, da der Entwurf der Nest-Operationen beide Paradigmen und die darin vorkommenden Betriebsarten auf uniforme Weise unterstützt; der Unterschied wird lediglich im Wert des transfer\_size-Attributs angezeigt.

Normalerweise dienen Geräte-Treiber zum Transfer von Datenblöcken auf Peripheriegeräte und brauchen daher nur die Operationen statischer Nester und nur singleuser-Kompetenzen<sup>4</sup> zu implementieren.

Eine mindestens multiuser-fähige Sonderform ist device\_mem, die transienten Speicher ohne transfer-Operationen zur Verfügung stellt, indem sie die Operationen get\_address- und put\_address implementiert. Um Speicher mit verschiedenen transfer\_size-Werten verwalten zu können, bilden mehrere Instanzen von device\_mem eine miteinander verdrahtete Hierarchie, bei der sich Verwalter kleiner Granularität den Speicher über einen Eingang bei Verwaltern mit größerer Granularität „ausleihen“ (vgl. Abschnitt 5.1). Der „Urverwalter“ des physischen Hauptspeichers besitzt den gesamten Speicher beim Start des Betriebssystems und hat daher keinen Eingang<sup>5</sup>.

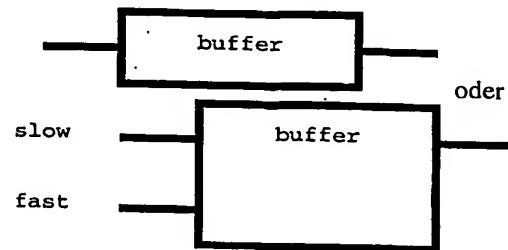
<sup>3</sup>Dies ist eine konzeptuell vereinfachende Darstellung. Konkrete Hardware besitzt oft eine innere Hierarchie, etwa Peripherie-Busse wie PCI oder Geräte-Busse wie SCSI. Zur Realisierung deren (Unter-)Treiber eignen sich prinzipiell ebenfalls die Abstraktionen Nest und Baustein, so dass der „eigentliche“ Geräte-Treiber auch einen oder mehrere Eingänge haben kann, mit denen er an die weitere Unter-Treiber-Infrastruktur angeschlossen wird. Weiterhin möchte ich darauf hinweisen, dass sich auch Hardware-Konzepte wie Memory-Mapped-IO, Grafikkarten-Framebuffer und dergleichen leicht und effizient durch die Abstraktion der Nester darstellen lassen.

<sup>4</sup>Auf dem Gebiet der Storage Area Networks (SAN) könnte die Einführung multiuser-fähiger Treiber durchaus Vorteile bringen, insbesondere für höhere Redundanz und Ausfallsicherheit sorgen, eventuell auch die Last besser verteilen. Allerdings muss dazu das Verhalten der restlichen Komponenten der beteiligten Betriebssysteme mitspielen.

<sup>5</sup>Eine Hardware-Architekturen wie z.B. die IBM z-Serie erlauben den Austausch von Speichermodulen zur Laufzeit. Dies lässt sich dadurch modellieren, dass der Urverwalter die Anzahl seiner Ressourcen ändert. Dazu muss er notfalls bereits vergebene Ressourcen wieder zurückfordern; siehe Kapitel 5.

Eine weitere multiuser-fähige Sonderform ist device\_ramdisk, die eine limitierte Persistenz implementiert, die sich nicht über Stromausfälle hinweg erstrecken muss<sup>6</sup>.

#### 4.1.2 buffer



Ein buffer-Baustein sorgt für die Entkopplung von Aktivitäten zwischen Eingang und Ausgang (in beide Richtungen). Er eignet sich zur Adaption des *zeitlichen Zugriffsverhaltens* zwischen langsamem und schnellen Baustein-Instanzen; im Idealfall sollte er die einzige Stelle im Gesamtsystem darstellen, die die Probleme der sogenannten „Speicherlücke“ zu lösen hat.

Ein häufiger Anwendungsfall ist die Nachschaltung hinter ein device\_\*. Dazu sind oft nur die Operationen statischer Nester erforderlich, und der Eingang braucht nur singleuser-Verhalten zu zeigen; auch der Ausgang braucht nur singleuser-Kompetenz bereitzustellen, da im häufigsten Anwendungsfall nur eine einzige map\_\*-Instanz nachgeschaltet wird. Es gibt aber auch Anwendungen, bei denen mindestens multiuser-Verhalten und -Kompetenz benötigt wird, insbesondere die Nachschaltung hinter einem remote-Baustein (siehe Abschnitt 4.1.12).

Bei der Implementierung des multiuser-Verhaltens tritt das in der Literatur bekannte Problem der *Cachekohärenz* (vgl. [AB86, LH89, HP95]) mit einer vorgesetzten Instanz auf. Durch die in der Nest-Schnittstelle vorgenommenen notify\_\*-Operationen (vgl. Kapitel 5) ist dies vergleichsweise leicht lösbar.

Wenn man buffer-Bausteine als die einzige Stelle im Gesamtsystem ansieht, die die Entkopplungs-Problematik des *zeitlichen Zugriffsverhaltens* löst, und zwar *sehr effizient* löst, dann kann man auch noch einen Schritt weitergehen: man kann fast alle anderen Bausteine intern *statuslos* implementieren, oder zumindest weitgehend *statuslos*. Ein Baustein ist *statuslos*, wenn er zu jedem Zeitpunkt, in dem sich kein logischer Kontrollfluss in ihm aufhält, destruiert und anschließend erneut konstruiert werden kann, ohne daß dadurch ein geändertes Verhalten von außen sichtbar wird.

Statuslosigkeit ermöglicht eine deutliche Reduktion der inneren Komplexität vieler Bausteine im Vergleich zu konventionellen Implementierungen, weil die Verantwortung

<sup>6</sup>Eine mögliche Verklemmung des Gesamtsystems durch im Normalbetrieb nicht zurückforderbare (quasi-persistente) Datenblöcke muss vermieden werden. Eine einfache Lösung dafür ist die Forderung, dass die Summe der Maximallängen aller device\_ramdisk-Instanzen den verfügbaren Hauptspeicher nicht überschreiten darf. Alternativ dazu kann auch das Verfahren von Habermann [Hab69] angewandt werden, mit dem eine Überbuchung der vorhandenen Hauptspeicher-Ressourcen prinzipiell möglich ist (jedoch wegen des damit verbundenen Wartens unakzeptable Auswirkungen auf das von Benutzern erwartete Verhalten von interaktiven Anwendungen oder von Realzeit-Anwendungen haben kann).

zur korrekten Aufbewahrung der internen Zustandsinformation an einen untergeordneten Baustein delegiert wird; daher ist Statuslosigkeit hochgradig erstrebenswert. Sie setzt allerdings voraus, dass Zugriffe über eine dermaßen entkoppelte Schnittstelle *fast nichts kosten*. Durch standardmäßige Benutzung von Prozeduraufrufen als Schnittstellen-Mechanismus und durch die Zero-Copy-Architektur (vgl. Abschnitt 2.8.2) wird dies ermöglicht.

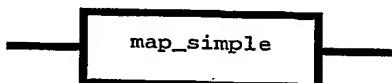
Zum Caching von Datenblöcken wird der Hauptspeicher genutzt; dieser muss von mehreren *buffer*-Instanzen geteilt werden. Die damit verbundene Problematik könnte außerhalb des Verdrahtungs-Systems der Bausteine gelöst werden (erstes Bild). Die folgende Lösung hat jedoch Vorteile:

Der mit *slow* bezeichnete Eingang soll wie bei der ersten Variante zur Entkoppelung der Zugriffs-Häufigkeiten und -Anzahlen dienen. Das mit *fast* bezeichnete Eingangs-Nest enthält den *gesamten Status* des Puffers; dazu gehört neben den Inhalten der zu puffernden Datenblöcke auch die transiente Zuordnung zwischen logischen und physischen Adressen und der Aktualitäts-Status. Der Baustein wird dadurch selbst vollkommen status-

Damit dies zu guter Performanz führt, muss *fast* sehr schnellen Zugriff bieten. Die Idee besteht darin, an diesem Eingang wahlweise ein *device\_ramdisk* anzuschließen, oder irgendein anderes relativ schnelles Gerät; bei der Pufferung von Zugriffen auf extrem langsame Bandlaufwerke<sup>7</sup> kann dies beispielsweise auch ein Nest sein, das auf Festplatte vorgehalten wird. Letztlich macht *buffer* nichts anderes, als das zeitliche Zugriffsverhalten des *fast*-Eingangs an den Ausgang weiter zu reichen. Die am *fast*-Eingang angeschlossene Instanz wird mit geringeren Datenmengen belastet als beim *slow*-Eingang vorhanden sind; bei Speicher-Defizit darf sie Speicher-Rückforderungs-Anträge (vgl. Kapitel 5) stellen.

Die explizite Benutzung von *device\_ramdisk* als *fast*-Eingang hat den weiteren Vorteil, dass die Probleme der Speicher-Rückforderung, sowie der im Gesamtsystem gemischt verwendeten unterschiedlichen *transfer\_size*-Blockgrößen ausschließlich dort zu lösen sind. Und von absehbaren Fortschritten in der Hardware-Entwicklung ist zu erwarten, dass zukünftige Rechner ganz andere interne Speicher-Hierarchien besitzen werden, als sie heute üblich sind. Das Baustein-Konzept ermöglicht eine flexible Anpassung an geänderte Rahmenbedingungen.

#### 4.1.3 map\_\*



<sup>7</sup>Früher galt die Grundregel: je mehr Kapazität ein Peripheriegerät bereitstellt, desto langsamer ist es. Zum Zeitpunkt der Niederschrift dieser Arbeit haben billige als Massenprodukte hergestellte Festplatten die 100-GByte-Grenze durchbrochen und damit die gleiche Größenordnung wie Bandlaufwerke erreicht (vielleicht noch mit Ausnahme weniger sehr teurer High-End-Modelle). Auch die Medienkosten pro GByte nähern sich langsam derselben Größenordnung. Ich erwarte, dass sich ähnliche Verschiebungen der momentan herrschenden Speicher-Hierarchien auch in Zukunft ereignen werden.

Aufgabe dieses Baustein-Typs ist, ein statisches Nest in ein dynamisches umzuwandeln. Es gibt daher nur einen Eingang (der meist nur *singleuser*-Verhalten<sup>8</sup> zu implementieren braucht) und einen Ausgang, der im Falle von Netzwerk-Betriebssystemen mindestens *multiuser*-Kompetenz bereitstellen sollte; die Herstellung dieser Kompetenz kann aber auch an eine nachgeschaltete oder interne *adaptor\_\**-Instanz (siehe Abschnitt 4.1.8) delegiert werden.

Es lassen sich verschiedene Arten von *map\_\**-Bausteinen realisieren, die ihre Aufgabe mit jeweils anderen internen Realisierungsverfahren lösen und an spezielle Last- und Benutzungsmodelle angepasst sind. Als Beispiele werden nun zwei mögliche Realisierungen beschrieben, *map\_simple* und *map\_simple\_delta*.

Die Umwandlungs-Aufgabe wird speziell bei *map\_simple* folgendermaßen erledigt: es wird angenommen, dass die Maximalgrößen des Eingangs- und Ausgangs-Nestes annähernd gleich sind, und dass der Konsument am Ausgang nur Anforderungen in Blockgrößen stellt, wie sie von der MMU für die virtuelle Speicherverwaltung gestellt werden (typischerweise 4KByte oder 8KByte). Das *transfer\_size*-Attribut des Ausgangs-Nestes wird also auf eine derartige Blockgröße gesetzt. Dies bedeutet, dass alle Operationen, insbesondere auch die *move*-Operation, nur in Vielfachen dieser Transfergröße erfolgen dürfen. Damit liegt eine effiziente Realisierung bereits auf der Hand: man benutze eine Tabelle, die eine Abbildung der Blocknummern bzw. -Adressen des Ausgangs-Nestes auf diejenigen des Eingangs-Nestes realisiert. Diese Tabelle ist größenordnungsmäßig etwa um den Faktor 1000 kleiner als die Größe des umzuwandelnden Nestes, verursacht also einen Platz-Overhead von etwa einem Promille, der sinnvollerweise am Anfang des Eingangs-Nestes vorreserviert wird. Alle Operations-Aufrufe, die etwas mit den virtuellen Adressen zu tun haben (get und ggf. *lock* / *unlock*) und die vom Ausgang her ankommen, werden nach Übersetzung durch diese Tabelle an den Eingang durchgereicht. Die *move*-Operation wird durch eine Verschiebung innerhalb der Tabelle realisiert, die aus den bereits dargestellten Gründen etwa um den Faktor 1000 weniger kostet, als wenn man *move* durch Unmengen von Block-transfer- und Kopieroperationen im statischen Eingangs-Nest realisiert hätte (was z.B. ein Baustein mit dem Namen *map\_braindead* besorgen könnte). Diese Eigenschaft lässt sich dazu ausnutzen, um den Aufwand weiter zu senken: wenn man die Tabelle wiederum in einem eigenen privaten Nest realisiert, kann dieses wiederum mit Hilfe einer beliebigen anderen *map\_\**-Instanz verwaltet werden, die lediglich die *transfer\_size*-Granularität der Tabellenelemente unterstützen sollte (beispielsweise *map\_simple\_delta*, *map\_braindead*, oder eine Kombination aus *map\_simple* mit einem nachgeschalteten *adaptor\_\**, der in Abschnitt 4.1.8 beschrieben wird). Bei günstiger Wahl und Kombination der Bausteine zu mehrstufigen Kaskaden lassen sich damit die Kosten

<sup>8</sup>Die Herstellung von *multiuser*-Eingängen ist möglich, aber je nach gewählter interner Struktur u.U. aufwendig. Auch bei Client-Server-Anwendungen ist es sinnvoll, ein statisches Nest erst mittels *map\_\** in ein dynamisches umzuwandeln, bevor es z.B. mittels *remote* exportiert wird.

insbesondere bei sehr großen Nestern nochmals um einige Zehnerpotenzen senken.

Eine Adressübersetzung verursacht nach längerem Betrieb mit vielen `move`-, `clear`- und `delete`-Operationen ein Phänomen, das in der Literatur über Dateisysteme als *Fragmentierung* (vgl. z.B. [MJLF84, McK96]) bekannt ist. Dieses Phänomen beschreibt die Tatsache, dass eine hohe Lokalität von Zugriffen auf der Ausgangsseite des map-Bausteins nicht unbedingt in eine hohe Lokalität der Zugriffe auf der Eingangsseite übersetzt wird, da die Tabelle die Adressen ähnlich wie bei Hash-Verfahren kräftig durcheinander würfeln kann. Ein Konsument erwartet jedoch, dass Schreib- oder Leseaufträge, die in zusammenhängender und aufsteigender Adress-Reihenfolge gegeben werden, mit deutlich besserer Performanz ausgeführt werden als zufällig verteilte Zugriffe<sup>9</sup>. Zur Lösung dieses Problems stehen mehrere bekannte Verfahren zur Verfügung, die aus der Literatur über Dateisysteme analog übertragbar sind, beispielsweise intelligente Allokations-Strategien und Defragmentierungs-Läufe. Durch das Background-IO-Konzept (Abschnitt 2.8.3) wird insbesondere eine Defragmentierung im Hintergrund des laufenden Betriebes stark erleichtert, ohne die Abarbeitung der Benutzeraktivitäten merklich zu stören. Ich erwarte, dass die Anwendung ganz einfacher und primitiver Verfahren wie die beständig angestrebte Herstellung einer Identitäts-Abbildung oder „Beinahe-Identitäts-Abbildung<sup>10</sup>“ sehr gute Resultate liefern wird, *sofern dies im Hintergrund* mittels Background-IO geschieht.

Wenn man davon ausgehen kann, dass eine Defragmentierung im Hintergrund dafür sorgt, dass die Adress-Übersetzung „nur wenig“ von einer Identitäts-Abbildung abweicht, dann lässt sie sich auch auf folgende Weise realisieren, die ich `map_simple_delta` nenne:

Statt einer Tabelle fester Größe wird eine Liste (bzw. ein als Ringpuffer ausgeführtes Array oder dergleichen) der bisher ausgeführten Operationen benutzt, die etwas an den Adress-Zuordnungen ändern. Dadurch geht die Durchführung von `move`, `clear` und `delete` rasant schnell, weil nur ein Eintrag in die Liste gemacht werden muss, so dass sich im Endeffekt ein *Log* der durchzuführenden Operationen ergibt (falls im Log auch alle IO-Operationen aufgezeichnet werden, entsteht ein ähnlicher Effekt wie in Log-strukturierten Dateisystemen, vgl. [RO91]). Die Übersetzung einer Adresse kann nun grundsätzlich dadurch erfol-

<sup>9</sup>Diese Erwartungshaltung ist bei genauer Betrachtung letztlich nur durch die Tatsache gerechtfertigt, dass die heutige Externspeicher-Technik, vor allem diejenige von Festplatten, schlechte Lokalitätseigenschaften aufgrund der mechanisch bewegten Teile besitzt (siehe auch die in der Literatur ausführlich diskutierte so genannte „Speicherlücke“). In absehbarer Zukunft werden hoch-kapazitive Externspeicher mit nicht nur deutlich höheren Datentransferraten und geringeren Latenzen, sondern auch besseren Lokalitätseigenschaften zu konkurrenzfähigen Preisen verfügbar sein, beispielsweise holographische Speicher oder auf Magnet- oder Quanteneffekten beruhende hoch-kapazitive Halbleiterspeicher. Aus diesem Grund dürfte die Diskussion zur Behebung des Fragmentierungs-Problems nur begrenzte Bedeutung haben.

<sup>10</sup>Hierunter fallen z.B. Verschiebungen um konstante Offsets, wie sie auf jeden Fall notwendig werden, wenn für die Übersetzungstabelle beispielsweise am Anfang des Eingangs-Nestes Platz reserviert wird. Ich möchte hier keine saubere mathematische Definition solcher Abbildungen geben, sondern eine Charakterisierung von Eigenschaften, die für den Architektur-Entwurf relevant sind.

gen, dass in dieser Log-Liste nachgesehen wird, ob irgendwelche Operationen eingetragen wurden, die etwas an einer gegebenen Adresse ändern; diese Änderungen werden dann in der richtigen Reihenfolge quasi virtuell nachvollzogen. Dieses einfache Verfahren führt natürlich nur dann zu akzeptabler Performanz, wenn die Liste möglichst kurz gehalten wird. Eine Möglichkeit zur Kürzung der Liste besteht darin, dass solche Operationen, die sich gegenseitig ganz oder teilweise aufheben bzw. die sich durch eine einzige Ersatz-Operation mit gleicher Semantik ersetzen lassen, aus der Liste entfernt und ggf. durch die Ersatz-Operation ersetzt werden (Prinzip der *Kompensation* von Operationen). Eine andere, in der Praxis auf Dauer unumgängliche Möglichkeit besteht darin, dass im Hintergrund Verschiebungen und Umordnungen durch Background-IO stattfinden, durch die Einträge in der Liste überflüssig werden<sup>11</sup>.

Zu erwähnen ist weiterhin, dass `map_simple_delta` nicht nur Blöcke mit relativ großer Granularität wie bei Platten-Devices oder MMU-Seiten verwalten kann, sondern auch kleinere `transfer_size`-Werte bis hinunter zu 1 unterstützt. Der Ausgang unterstützt automatisch die gleiche Granularität, wie sie vom Eingang vorgegeben wird, allerdings kann es vorkommen, dass eine get-Anforderung mit einem hohen Vielfachen der `transfer_size` nur mit einem geringeren Vielfachen möglich ist. Dies ist in der Schnittstelle für Nester ausdrücklich so vorgesehen, und es ist die Aufgabe eines jeden Konsumenten, mit diesem Fall umgehen zu können (wobei er sich das Problem auch durch Vorschalten eines `adaptor_*` vom Hals schaffen kann).

Ein `map_*`-Baustein muss auch die Operation `get_meta` sowie die Operationen auf dem Meta-Nest implementieren. Ein Meta-Nest sollte im Regelfall eine `transfer_size` von 1 unterstützen, was sich durch interne Verwendung eines `adaptor_*`-Bausteins (Abschnitt 4.1.8) erzielen lässt.

Wichtig ist ferner, dass `map_*`-Bausteine unbedingt die Eigenschaft der *Absturzfestigkeit* besitzen sollten. Mit diesem Begriff soll umschrieben werden, dass eine jederzeitige Trennung des Bausteins von seinem Eingang bzw. ein Verlorengehen von `transfer`-Operationen (z.B. bei einem plötzlichen Stromausfall, der zum Verlust der vorgesetzten flüchtigen `buffer`-Informationen führt) nicht die Integrität der Adressübersetzung zerstören darf. Wenn man nicht auf die Performanz-Vorteile von Pufferung in flüchtigem Speicher verzichten will, muss man damit leben, dass der Zustand auf dem persistenten Hintergrundmedium ständig dem Zustand im flüchtigen Speicher hinterherhinkt, so dass bei unerwartet auftretenden Störungen Daten und damit Informationen verloren gehen. Bekannte Lösungen dieses Problems<sup>12</sup> stellen Log-basierte Schreibver-

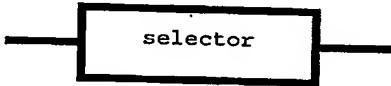
<sup>11</sup>Unabhängig davon sollte `map_simple_delta` nur bei solchen Last-Charakteristiken angewandt werden, bei denen nur relativ selten Verschiebe-Operationen angefordert werden (was bei vielen Anwendungen durchaus erwartet werden kann), oder aber bei denen umgekehrt extrem häufige Verschiebungen, dafür aber nur relativ wenige Adress-Zugriffe vorkommen, so dass der Adressübersetzungs-Aufwand bei den amortisierten Kosten nur wenig ins Gewicht fällt.

<sup>12</sup>Eine weitere Lösung, nämlich die Konsistenzprüfung nach Abstürzen, halte ich wegen ihrer schlechten Reparatur-Eigenschaften und vor allem wegen ihrer Komplexität für unpraktikabel (als Standard-Mittel zur Behebung der Effekte von Abstürzen; in absoluten Notsituationen sieht es dagegen anders aus). Die Komplexität einer Konsistenzprüfung kann we-

fahren dar; entweder in Form separater Logs oder in Form sogenannter Log-strukturierter Speicher. Zur Unterstützung dieser Verfahren wurde in Abschnitt 3.3.1 der Parameter `depend` eingeführt, mit dem sich die Reihenfolge von Schreiboperationen teilweise vorgeben lässt. Absturzfestigkeit lässt sich fernerhin als Teil von ACID-Transaktionen implementieren, falls man diese bereits in einer entsprechenden `map_*`-Variante bereitstellen möchte. Eine weitere Möglichkeit stellt die Vorschaltung eines eigenen `power-safe`-Bausteins analog zu [dJ93] dar, der in isolierter Weise nur die Absturzfestigkeit als Strategie implementiert und auf die Erweiterung statischer Nester auf dynamische verzichtet.

Die Funktionalität von `map_*` lässt sich sicher auch mit Dutzenden weiterer Verfahren zur Adress-Übersetzung herstellen, die eventuell Vorteile bei bestimmten Last-Charakteristiken bringen können. Zu nennen sind hier beispielsweise Tries, balancierte Bäume, B-Bäume, Fuzzy Hashing, und andere. Meine Ausführungen sollten plausibel gemacht haben, dass effiziente Realisierungen des Konzeptes der dynamischen Nester möglich sind.

**selector**

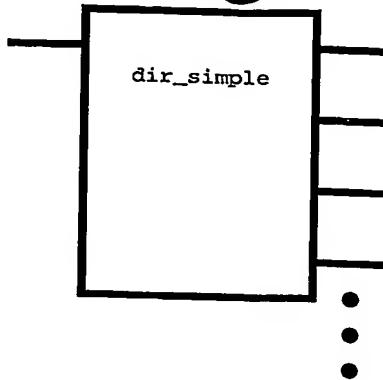


Dieser Anpassungs-Baustein schneidet einen zusammenhängenden Teil-Adressraum aus seinem Eingangs-Adressraum aus und stellt ihn unverändert am Ausgang zur Verfügung, wobei die Adressen standardmäßig wieder bei 0 neu beginnen. Wird der ausgeschnittene Adressraum als Ganzes verschoben, dann wirkt sich das nicht auf den Gastgeber-Adressraum aus, ebenso umgekehrt bei Überstreichung einer move-Operation im Gastgeber-Adressraum über den gesamten ausgeschnittenen Bereich (Transparenz).

Implementierung ist relativ einfach, da außer einer normalen Adressübersetzung und -Überprüfung nichts zu machen ist und alle Operationen vom Ausgang an den Eingang durchgereicht werden können. Es sind zwei grundlegende Varianten von Selektoren möglich, `selector_persistent` und `selector_tmp`, die sich darin unterscheiden, ob die Ausschnitts-Adressen und -Längen sowie deren Änderungen persistent festgehalten werden sollen oder nicht. Im ersten Fall wird Hilfsinformation im Meta-Nest des Eingangs gehalten, mit der die Startadresse und die Länge relativ zum Gast-Adressraum auf Dauer festgehalten wird, so dass eine Destruktion des `selector` mit anschließender Reinstantiierung wieder den alten Zustand ergibt.

#### 4.1.5 `dir_*`

gen der möglichen Zersplitterung eines Nestes in sehr viele kleine Flicken nach Art eines Schweizerkäses sehr groß werden.



Ein Baustein dieser Art benutzt das Eingangs-Nest als eine Art Sammel-Container, um mehrere von einander unabhängige Ausgangs-Nester daraus zu bilden. Die Ausgangs-Nester werden bei Bedarf durch die baustein-spezifische Zugriffsoperation `create` neu erstellt; bereits früher erstellte werden durch `lookup` erneut instantiiert, sofern sie nicht bereits instantiiert sind. Unterschieden werden die verschiedenen möglichen Ausgangs-Instanzen durch einen Suchschlüssel, der als ein zusammenhängender Datenblock übergeben wird. Zur Abfrage aller vorhandenen persistent gespeicherten Schlüsselwerte dient ein spezieller Ausgang, der Verzeichnis-Nest genannt wird, der im Regelfall eine `transfer_size` von 1 unterstützt, und von dem nur gelesen werden darf. Ein Verzeichnis-Nest stellt ein Nest mit lückenlos liegenden Datenpaketen dar, wobei jedes Paket genau einen der vorhandenen Suchschlüssel enthält. Die Schlüsselwerte können, müssen aber nicht nach irgendeinem internen Kriterium sortiert gehalten werden. Löschen von Schlüsselwerten und der zugehörigen Ausgangs-Nest-Instanzen lassen sich als baustein-spezifische Operation und/oder durch eine `delete`-Operation auf dem Verzeichnis-Nest realisieren. Hierbei ist die aus Unix bekannte Methodik zu bevorzugen, dass zwischen statischen und dynamischen Referenzen unterschieden werden sollte, und dass ein Ausgang und sein Speicherplatz erst dann tatsächlich freigegeben werden sollen, wenn keine dynamischen Referenzen mehr vorhanden sind.

Aus dieser Beschreibung dürfte klar geworden sein, dass damit ein Verzeichnis realisierbar ist, wie es in konventionellen Betriebssystem-Architekturen von Dateisystemen zur Verfügung gestellt wird. Ein `dir_*` stellt jedoch keine Verzeichnisbaum-Hierarchie zur Verfügung, sondern ähnelt eher dem flachen Index einer Datenbank. Dennoch lassen sich Verzeichnisbaum-Hierarchien sehr leicht herstellen: am Ausgang einer `dir_*`-Instanz braucht lediglich eine weitere `dir_*`-Instanz angeschlossen zu werden und so weiter. Auf diese Weise kann der Dateisystem-Baum (bzw. ein momentan instantiiertes Teilbaum davon) direkt als Baumstruktur von Baustein-Instanzen mit der zugehörigen Verdrahtung dargestellt werden. Wenn man die in Abschnitt 4.2 vorgestellte Auto-Instantiierung von Bausteinen benutzt, dann braucht man sich als Benutzer nicht um die dynamische Herstellung dieser Baumstruktur zu kümmern, da dies automatisch geschieht.

Im Vergleich zur Funktionalität klassischer Dateisysteme ermöglicht dieses Konzept eine wesentlich flexiblere Instantiierung und Verdrahtung, da man beispielsweise

- für jedes Verzeichnis individuell verschiedene Baustein-Typen einsetzen kann, beispielsweise `dir_simple` für kleine Verzeichnisse, `dir_hash` für solche mit besonders schneller lookup-Funktionalität, oder `dir_btreet` für solche mit besonders guten Lokalitätseigenschaften trotz riesiger Ausdehnung
- Datenkomprimierungs-, Datenverschlüsselungs- und sonstige Anpassungs-Bausteine wie `adaptor_*` und andere auf beliebigen Hierarchieebenen (automatisch) dazwischen schalten kann
- auf Konzepte wie Mounts und Mount-Tabellen verzichten kann, da dies von einem `redirect`-Baustein übernommen werden kann
- ebenso auf Konzepte wie Loopback-Devices verzichten kann, indem lediglich ein `map_*` zwischengeschaltet wird
- nahtlose Integration mit der Funktionalität von Datenbanken möglich ist: dazu zählt nicht nur der später vorgestellte `transaction`-Baustein, sondern auch die Möglichkeit, spezialisierte Bausteine wie beispielsweise `dir_fixed_keysize` einzusetzen, bei denen eine Uniformität der Schlüssellängen zugunsten besserer Platzausnutzung erzwungen wird
- eine *virtuelle* Herstellung von Verzeichnisinhalten durchführen kann, beispielsweise mit einem Baustein `dir_proc` für die Funktionalität von `/proc`-Dateisystemen, oder `dir_join` zur Herstellung der Funktionalität der aus der Datenbank-Welt bekannten Join-Operation aus dem Daten-Inhalt mehrerer anderer Nester. In diesem Fall können `dir_*`-Varianten entstehen, die gar keinen oder mehrere Eingänge besitzen, was sich u.U. mit dem althergebrachten Konzept eines Dateisystem-Baums nicht auf intuitive Weise modellieren lässt.

Die grundlegende Funktionsweise von `dir_*`-Bausteinen möchte ich am Beispiel einer Realisierung von `dir_simple` erläutern. Es werden insgesamt drei verschiedene Regionen von zusammenhängenden Adressbereichen benutzt, um die Container-Funktionalität zu realisieren. Dies sind

- die Index-Region,
- die Meta-Region,
- die Daten-Region.

Diese Regionen werden innerhalb des Eingangs-Nestes und des Eingangs-Meta-Nestes angelegt, wobei die Daten-Region auf jeden Fall im Eingangs-Nest, die Meta-Region auf jeden Fall im Eingangs-Meta-Nest, und die Index-Region wahlweise in einem der beiden Nester liegen kann (sinnvollerweise sollte die Zuordnung von der Größe des Index abhängig gemacht werden).

`dir_simple` betrachtet diese Regionen jeweils als kompakt zusammenhängende Teil-Nester und benutzt die

am Eingang zur Verfügung stehenden `move`-Operationen, um Einfügungen und Löschungen von Index-Werten durchzuführen, sowie die an den Ausgängen ankommenden `move`-Operationen u.a. an den Eingang weiter zu reichen. Dies bedeutet für die `get_maxlen`-Werte der Ausgänge, dass ihre Summe kleiner-gleich des `get_maxlen`-Wertes des Eingangs sein muss<sup>13</sup>. Die interne Realisierung von `dir_simple` braucht sich nicht um die Bereitstellung und Verschiebung von Platz zu kümmern, da diese Funktionalität bereits am Eingang verfügbar ist. Um die Funktionalität eines instantiierten Ausgangs zu erfüllen (mit Ausnahme von `set_maxlen`, das abgefangen werden muss), können interne Instanzen von `selector` verwendet werden.

Die Reihenfolge von Einträgen ist bei `dir_simple` in allen Regionen gleich: wenn beispielsweise ein Eintrag in der Index-Region ganz am Anfang bei der ersten Position eingefügt wird, dann wird in der Meta-Region und in der Daten-Region ebenso verfahren und dort jeweils Platz im Adressraum geschaffen. In diesem Fall, oder wenn sich beispielsweise die Größe eines Ausgangs-Nestes durch `set_maxlen` so stark ändert, dass Überschneidungen drohen, wird der gesamte restliche Adressraum in einem Rutsch verschoben; wegen der Transparenz-Eigenschaft der intern verwendeten `selector`-Bausteine merken die bereits instantiierten Ausgänge nichts davon.

Der Index-Bereich wird bei `dir_simple` direkt im gleichen Format abgespeichert, wie sie der Verzeichnis-Nest-Ausgang verlangt. Andere `dir_*`-Arten können selbstverständlich davon abweichen.

Die Ausgänge eines `dir_*`-Bausteins sollten mindestens multiuser-Kompetenz bereitstellen. Hierfür gibt es mehrere Möglichkeiten: die interne Realisierung könnte z.B. nur durch `singleuser`-Verhalten erfolgen (was den Entwurf ein klein wenig vereinfacht) und die multiuser-Kompetenz an den Ausgängen durch eine `adaptor_*`-Instanz zur Verfügung stellen. Für die Einsatzgebiete von Netzwerk-Betriebssystemen ist jedoch durchgehendes multiuser-Verhalten vorteilhaft. Dies ist bei `dir_simple` relativ einfach zu erfüllen, da die oben beschriebene Realisierung weitgehend *statuslos* erfolgen kann. Bei statuslosen Realisierungen ist die Hinzunahme von multiuser- oder multiversion-Verhalten einfacher, da lediglich alle potentiell konfliktträchtigen Operationen durch `lock` / `unlock`-Paare zu klammern sind. Die einzige konfliktträchtige Operation bei den Ausgangs-Nestern ist `set_maxlen`, die bei vernünftigem Entwurf nur selten aufgerufen werden sollte; alle anderen Operationen außer Modifikationen in der Index-Region brauchen keine zusätzlichen Klammern und laufen unverändert oder nur wenig verändert durch.

<sup>13</sup>Konzeptionell entspricht diese Einschränkung dem auch bei konventionellen Dateisystemen vorkommenden Zwang, dass die Summe des Platzbedarfs aller abgespeicherten Dateien nicht die Gesamtgröße des Dateisystems übersteigen darf. Falls in einer `dir_*`-Hierarchie sehr große dünn-besetzte Nester vorkommen, deren Nutzungsgrad eine heuristisch bestimmte Schranke unterschreitet (was bei häufigem Vorkommen zum Problem führen kann, dass ein Adressraum mit 64 Bit Breite nicht mehr für die Summe aller vorkommenden Adressraum-Größen ausreicht), dann kann eine Verkleinerung der virtuellen Adressraum-Größen durch einen zwischengeschalteten `map_*`-Baustein erreicht werden.

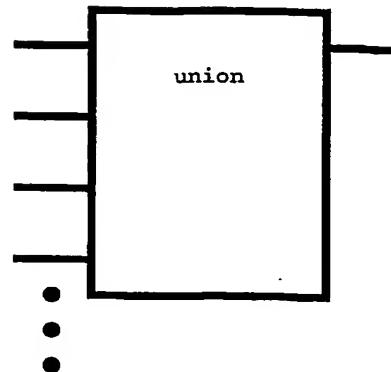
Diese recht einfache Realisierung hat sehr gute Lokalitätseigenschaften: das Lokalitäts-Verhalten eines Ausgangs wird direkt an den Container beim Eingang weitergereicht. Falls kleine Verzeichnisse so realisiert sind, dass die Index-Region im Meta-Nest des Eingangs liegt, dann enthält das Meta-Nest im Falle von rekursiv verschachtelten Verzeichnissen an der Wurzel die gesamte Baumstruktur aller Indizes, die im Vergleich zur Summe aller Dateigrößen um einige Größenordnungen kleiner ausfällt (zumindest bei üblichen durch menschliche Nutzer verursachten Dateigrößen) und allein wegen dieser geringen Lokalität der Ausdehnung einen schnellen Zugriff erlaubt (wobei das Caching des transiv vorgeschalteten buffer-Bausteins separate Konzepte wie Inode- oder Namens-Caches überflüssig macht).

Die Zuordnung der Index-Region zu einem der beiden Eingangs-Nester ist eine Strategie-Entscheidung, die in jedem Einzelfall anders getroffen werden kann; sehr große Index-Regionen lassen sich auch in das reguläre Eingangs-Nest verlagern, so dass die Lokalität des Meta-Nestes möglichst wenig verschlechtert wird.

Eine weitere strategische Entscheidungsmöglichkeit für Lokalitäts-Verhalten besteht darin, einige oder alle der Nester miteinander zu verschmelzen. Dies kann insbesondere bei der Verquickung der Index- mit der Meta-Region Vorteile bringen, sofern die Meta-Daten der Ausgänge nur wenig Platz beanspruchen. Der von Hans Reiser in *reiserfs* propagierte effiziente Zugriff auf sehr kleine Datei-Größen (wie z.B. bei der Modellierung einzelner Felder von Datensätzen einer Datenbank mit Hilfe von „Mini-Dateien“) lässt sich ggf. durch eine Verschmelzung von Index- und Daten-Region erreichen, bzw. durch eine fallweise dynamische heuristische Zuordnung zu einer der Regionen. Ferner kann man bei der Repräsentation der Felder von Datensätzen einer Datenbank einen funktionalitätsmäßig stark eingeschränkten Baustein-Typ *dir\_record* einsetzen, der die immer gleiche Index-Struktur von Records gleichen Aufbaus und die ebenfalls immer gleichen Metadaten (in denen die Feldnamen und Feldlängen und weitere Attribute gespeichert werden) nicht in der Index-Region jeder einzelner Instanz abspeichert, sondern per Reference einer gemeinsamen externen Quelle bezieht und damit für einen vernachlässigbaren Platz-Overhead bei jeder einzelnen Instanz verursacht.

Durch diese Beispiele sollte plausibel geworden sein, dass spezielle Anforderungen, die bisher oftmals als Anreiz zur Entwicklung aufwendiger und umfangreicher Dateisysteme mit trickreichen internen Implementierungen dienten, durch ein Baustein-Konzept mindestens ebenso gut abgedeckt werden können. Die hier vorgestellte Baustein-Zerlegung verteilt die in Dateisystemen vorkommenden Problematiken und Funktionalitäten auf mehrere Baustein-Arten, isoliert sie voneinander, und ermöglicht vorher unbekannte Kombinationen. Einige der Möglichkeiten werden auf einfachere Weise als mit stapelbaren Dateisystemen (vgl. [HP94, HP95]) gelöst, da nicht mehr zwischen verschiedenen Ebenen wie „Dateien“ versus „Dateisystem-Bäume“ unterschieden wird.

#### 4.1.6 union

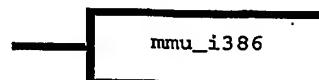


Dieser Baustein stellt in gewisser Hinsicht eine inverse Operation zu *selector* bzw. *dir\_\** dar: mehrere Eingangs-Nester erscheinen in einem Ausgangs-Nest und werden dabei adressmäßig nebeneinander aufgereiht. Über Parameter bzw. Meta-Informationen wird festgelegt, ob sich ein Eingangs-Adressraum lückenlos an seinen Vorgänger anschließen soll (so dass eventuelle Lücken am Ende des Vorgängers und am Anfang des eigenen Nestes zusammen geschoben werden und move-Operationen des Vorgängers mit vollzogen werden), ob der Anschluss über *get\_maxlen* erfolgen soll, oder ob er ggf. unter Lückenbildung an festen Adressen „festgenagelt“ erscheinen soll (analog zu *selector*). Weitere Spielarten sind denkbar.

Über den später besprochenen Mechanismus der generischen Operationen lassen sich Eingänge zur Laufzeit hinzufügen und entfernen.

Ein wichtiges Anwendungsgebiet von *union* ist die Zusammenstellung verschiedener Regionen bzw. „Segmente“ in virtuellen Adressräumen oder Schutzbereichen, beispielsweise die Einteilung in Code-, Stack- und Datensegmente, sowie in Mappings anderer Nester. Verschiedene Mapping-Arten lassen sich durch Vorschalten von Anpassungsbausteinen wie *cow* (Abschnitt 4.1.10) realisieren.

#### 4.1.7 mmu\_\*



Dieser Baustein besitzt genau einen Eingang und keinen Ausgang<sup>14</sup>, ist also im Sinne der Verdrahtungslogik ein reiner Konsument. Er stellt die Schnittstelle zur Memory-Management-Unit (MMU) der Rechner-Hardware dar. Das Nest wird hierbei 1:1<sup>15</sup> in einen virtuellen Adressraum umgewandelt, der sich über Kontrollflüsse direkt durch Maschinenbefehle des Prozessors adressieren lässt.

<sup>14</sup>In Wirklichkeit hat er auch einen Ausgang, nämlich den virtuellen Benutzer-Adressraum. Dieser hat jedoch aus Hardwaregründen nicht mehr dieselbe Schnittstelle mit den hier vorgestellten Grundoperationen. Im abstrakten Sinne führt ein Prozessor-MMU-Gespann dennoch die gleichen Grundoperationen aus: Hauptspeicher-Zugriffe des Prozessors lassen sich als Kombinationen von get- und put- mit transfer-Operationen modellieren; eine MMU betreibt abstrakt gesehen eine eingeschränkte Art der Virtualisierung, wie sie ein dynamisches Nest bereitstellt.

<sup>15</sup>Falls der logische Adressraum eines Nestes größer als der virtuelle Adressraum eines Prozesses ist (z.B. bei 32Bit-Hardware), dann wird nur ein Teil des logischen Nest-Adressraums für den direkten Zugriff durch Maschinenbefehle umgesetzt.

Die Realisierung dieses Bausteins ist im Vergleich zu konventionellen Implementierungen relativ einfach: wenn ein Seitenfehler auftritt, wird der betroffene Datenblock mittels `get` und nachfolgendem `transfer` im `read`-Modus vom Eingangs-Nest angefordert und in die Seiten-Tabellen<sup>16</sup> der MMU eingetragen. Beim Austragen wird `put`, im Falle geänderter Seiten davor auch noch `transfer` im `write`-Modus mit geringer Hintergrund-IO-Priorität aufgerufen.

Über die später besprochene `notify_*`-Schnittstelle erhält `mmu_*` Kenntnis von durch die Speicherverwaltung zur Freigabe vorgesehenen physischen Datenblöcken, sowie von beispielsweise durch `move`-Operationen invalide gewordenen Adressbereichen und trägt sie wie soeben besprochen aus<sup>17</sup>. Hierauf können sie evtl. beim nächsten Seitenfehler gleich wieder angefordert werden, was aber relativ geringe Verzögerungen zur Folge hat, *sofern* sich die Seite in Wirklichkeit noch im zentralen LRU-Cache der transitiv vorgeschalteten `buffer`-Instanz befindet. Solche Rückforderungen können daher *auf Verdacht* und jederzeit von der Speicherverwaltung an zentraler Stelle<sup>18</sup> ausgelöst werden, ohne die Aktivitäten eines Benutzerprozesses merklich zu stören. Dadurch ergibt sich eine fortlaufende Altersbestimmung innerhalb der Working-Sets (vgl. [Den68, Den71]) aller vorhandenen virtuellen Adressräume, was dem Effekt des bekannten Second-Chance-Algorithmus zur Seitenersetzung ähnelt. Als Nebeneffekt werden geänderte Seiten frühzeitig auf Verdacht mit Hintergrund-Priorität ausgelagert<sup>19</sup>, so dass bei einer später tatsächlich eintretenden Speicherknappheit eine Chance besteht, dass die Seite inzwischen *nicht wieder* modifiziert wurde und daher sofort recycelt werden kann, ohne auf die Beendigung von IO warten zu müssen.

In einem `mmu_*`-Baustein braucht keine wie auch immer geartete spezielle Paging-Strategie mit verschiedenen Modi und Abhängigkeiten von Mapping-Arten implementiert zu werden, was bei konventionellen Implementierungen den Löwenanteil an Komplexität ausmacht. Diese Aufgaben einschließlich der Persistenzhaltung privater Mapping-Segmente werden von vorgeschalteten Bausteinen übernommen<sup>20</sup>.

<sup>16</sup>Öblicherweise benötigt MMU-Hardware einen Hilfspeicher für die Seitentabellen. Dieser ist hier nicht als eigener Eingang ausgeführt, da eine andere Verdrahtung als mit `device_mem` aus Hardwaregründen meist nicht möglich ist.

<sup>17</sup>Trotz der Notwendigkeit, Hilfspeicher für die Seitentabellen einzusetzen, arbeitet eine `mmu_*`-Instanz logisch gesehen *statuslos*: man kann jederzeit die gesamte Seitentabelle freigeben, ohne dass ein Schaden (außer evtl. Performanz-Verschlechterung) eintritt, da die danach unweigerlich auftretenden Seitenfehler den Status der Tabelle wieder aus dem Eingangs-Nest soweit notwendig rekonstruieren werden.

<sup>18</sup>Nur diese Stelle hat die volle Übersicht über die momentane Speichersituation und kann einigermaßen verlässlich einschätzen, wie hoch zukünftige Speicher-Anforderungsraten ausfallen könnten oder ob die Situation unkritisch ist. Die Rückforderungen auf Verdacht lassen sich damit den aktuellen Lastverhältnissen anpassen, so dass unnötiger Overhead minimiert wird.

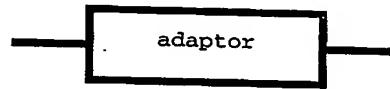
<sup>19</sup>Im Vergleich zu konventionellen Strategien, die mit der Auslagerung oftmals erst bei bereits eingesetzter Speicherknappheit beginnen, führt dies zu potentiell erhöhtem IO-Verkehr. Da dieser jedoch mit Hintergrund-Priorität abgewickelt wird, stört er laufende Aktivitäten so gut wie gar nicht.

<sup>20</sup>Ein konventionelles Paging auf Hintergrundspeicher stellt bei der hier vorgestellten Architektur kein eigenständiges Konzept dar. Private Seg-

Damit ein `mmu_*`-Baustein am Ende einer Baustein-Hierarchie stehen darf, muss die übliche Trennung in Spinlocks und schedulende Locks aufgehoben werden (siehe [K+91, LA93] sowie Abschnitt 3.3.5). Dies ist in SMP-Rechnern (Symmetric MultiProcessing) notwendig, um mehrere `mmu_*`-Instanzen auf verschiedene Prozessoren verteilen zu können oder mehrere auf verschiedene Prozessoren verteilte Kontrollflüsse auf der gleichen `mmu_*`-Instanz laufen zu lassen. Weiterhin ist erforderlich, dass Seitenfehler-Unterbrechungen die Nest-Operationen aufrufen dürfen (siehe [KE95] sowie Abschnitt 3.3.5).

Die Realisierung von Schutzbereichen ist ebenfalls Aufgabe von `mmu_*`. Dazu ist eine Verknüpfung mit der Kontrollfluss-Implementierung nötig, die am besten außerhalb der regulären Baustein-Verdrahtung gelöst wird, da sie davon unabhängig ist (dies gilt ebenfalls für die Anbindung an die Unterbrechungen)<sup>21</sup>. Verschiedene Schutzbereiche lassen sich am einfachsten durch Zuordnen verschiedener Mandate (vgl. Abschnitt 2.7) und unterschiedlicher Behandlung in untergeordneten `check_*`-Instanzen implementieren. Damit werden `mmu_*`-Instanzen zu Verwaltern derjenigen Mandate, die mit den Schutzbereichen zu tun haben.

#### 4.1.8 adaptor\_\*



Es handelt sich um einen Anpassungs-Baustein mit nur einem Eingang und einem Ausgang, der zwischen Nester mit verschiedenen Kompetenzen und Verhalten wie beispielsweise verschiedenen `transfer_size`-Attributwerten vermittelt und übersetzt.

mente werden stets als temporäre Nester ausgeführt, die auf Massenspeicher gehalten werden. Im Falle von Stack-Segmenten sind sie am Anfang (beinahe) leer; bei wachsender Größe wird neuer Platz mittels `clear` nachgefordert; die Auslagerung ihres Inhaltes kann spekulativ durch Background-IO erfolgen.

<sup>21</sup>Mitte der 1990er Jahre wurden im Fachgebiet der Betriebssysteme besonders viele Diskussionen um Kern-Größen und um Grenzen zwischen Kern und „Benutzerbereich“ geführt. Ich vermeide den Begriff eines Betriebssystem-Kerns, da er oft mit monolithischen Strukturen assoziiert wird. In der hier vorgestellten Architektur kann es *mehrere* Kerne im konventionellen Sinn geben: beispielsweise kann die Anbindung an die MMU-Hardware getrennt von der Kontrollfluss-Verwaltung erfolgen, ggf. auch in unterschiedlichen Schutzbereichen. Wenn von einem bestimmten Hardware-Mechanismus in einem System nur ein einziges Exemplar vorhanden ist (beispielsweise eine gemeinsame Unterbrechungs-Sprungtabelle in einem Multiprozessor-System), dann ergibt sich für die zugehörige Verwaltungs-Software auf natürliche Weise nur eine einzige Verwaltungs-Instanz. Bei prinzipiell voneinander unabhängigen Hardware-Einheiten kann es jedoch durchaus mehrere verschiedene Verwaltungs-Instanzen geben, die man in konventioneller Terminologie mehrere voneinander unabhängige Mikro-Kerne nennen könnte. Eine derartige Modularisierung bringt insbesondere bei der Anpassung an geänderte oder erweiterte Hardware-Komponenten Vorteile. Dieses Problematisches tritt insbesondere durch die fortschreitende technische Entwicklung bei Gerätetreibern auf und wird bei vielen Minimalisierungs-Ansätzen von Kernen wie z.B. der Exokernel-Architektur [EKO95] nur unzureichend gelöst.

#### 4.1.8.1 Adaption zwischen verschiedenen Zugriffs-Modellen

Wie der Tabelle 3.1 auf Seite 22 zu entnehmen ist, gibt es folgende Fälle, in denen kein Anschluß von Eingängen an die Ausgänge anderer Bausteine möglich ist:

1. Der Ausgang hat zu geringe Kompetenzen, um die Anforderungen durch das Verhalten der Eingänge abdecken zu können.
2. Ein Eingang unterstützt nur `singleuser`-Verhalten und verträgt sich daher nicht mit anderen Eingängen.

Für jeden dieser Fälle schlage ich vor, eine `adaptor`-Art einzuführen, die das jeweilige Problem löst. Fall 1 lässt sich beispielsweise durch `adaptor_multi` lösen, der die am Eingang nicht vorhandenen Lock-Operationen nachimplementiert. Zur Herstellung der `multiuser`-Kompetenz brauchen die restlichen Operationen nur durchgereicht zu werden. Etwas aufwendiger ist die Herstellung der `multiversion`-Kompetenz, da in diesem Modus von jedem Datenblock potentiell mehrere Versionen ins System gebracht werden müssen und verwaltet werden müssen. In diesem Fall ist es sinnvoll, einen weiteren `tmp`- oder `fast`-Eingang analog zu `buffer` bzw zu `cow` (siehe Abschnitt 4.1.10) einzuführen, der die zusätzlichen Versionen und die damit verbundene Statusinformation enthält.

Zur Lösung des Falls 2 schlage ich vor, einen Baustein `adaptor_synchronize` einzuführen. Dieser kann einen Eingang mit `multiuser`- oder `multiversion`-Verhalten voraussetzen, da diese Funktionalität notfalls durch `adaptor_multi` bereitgestellt werden kann. Da an jedem Ausgang immer nur ein einziger Konsument mit `singleuser`-Verhalten angeschlossen werden darf, muß `adaptor_synchronize` entsprechend viele Ausgänge haben, die ggf. zur Laufzeit nachträglich instantiiert oder gelöscht werden müssen.

Die Problematik von `adaptor_synchronize` besteht darin, daß jeder Ausgang die Illusion erhalten muß, es handle sich um den einzigen Konsument einer Nest-Instanz, der Änderungen an dem Datenblock durchführt. Eine solche Illusion wird auch von klassischen Datenbank-Transaktionen erzeugt. Zur Herstellung dieser Illusion ist daher prinzipiell auch der Baustein `transaction` (Abschnitt 4.1.11) geeignet. Aus der Transaktionstheorie ist bekannt, dass eine vollständige Isolation der Teilnehmer bei beliebigen, *nicht vorhersagbaren* und *echt parallelen* Aktionen verschiedener `singleuser`-Teilnehmer nicht möglich ist, ohne die Gefahr von Deadlocks oder Rollbacks in Kauf zu nehmen. Daher schlage ich eine Aufteilung der Funktionalität vor: echte Parallelität soll durch `transaction` ermöglicht werden; eine eingeschränkte und leichter zu implementierende Form der Illusion von `singleuser`-Verhalten wird durch `adaptor_synchronize` hergestellt. Dies geschieht folgendermassen:

Wir nehmen an, daß die Konsumenten an den Ausgängen *statuslos* arbeiten, d.h. sie fordern Datenblöcke mittels `get` nur für einen relativ begrenzten Zeitraum an und geben sie möglichst bald wieder mittels `put` frei. Bei völiger Statuslosigkeit kann man damit rechnen, dass jeder

Konsument recht bald alle seine Blöcke wieder freigeben wird, oder dass zumindest eine vollständige Rückforderung mittels `notify_*`-Operationen (siehe Kapitel 5) möglich ist. Unter dieser Voraussetzung läßt sich eine einfache Strategie durch Locks implementieren, indem zu einem Zeitpunkt jeweils nur ein einziger Ausgang Zugriff auf den Status des Eingangs erhält; die anderen müssen solange warten, bis dieser sämtlichen Status zurückgegeben hat. Sobald irgendwelche Anforderungen durch einen anderen Ausgang ankommen, während irgendein Status bereits vergeben ist, dann versucht die `adaptor_synchronize`-Implementierung, den anderweitig vergebenen Status mittels `notify_*` wieder baldmöglichst zurückzubekommen<sup>22</sup>. Dies schränkt die Parallelität leider sehr stark ein, ist aber einfach zu implementieren, vermeidet Deadlocks und kommt ohne Rollback-Operationen aus, die bei echt parallelen Transaktionen und vorher nicht bekanntem Zugriffsverhalten nicht vermeidbar<sup>23</sup> sind.

Die Konsequenz aus dieser Misere ist meines Erachtens, dass man das `singleuser`-Programmiermodell vermeiden sollte und Konsumenten mit explizitem `multiuser`- oder `multiversion`-Verhalten<sup>24</sup> (Kapitel 7) ausstatten

<sup>22</sup>Dies zeigt eine gewisse Ähnlichkeit zu den *opportunistischen Locks*, die insbesondere von Microsoft eingesetzt werden [OpL]. Opportunistische Locks brauchen jedoch nicht unbedingt gewährt zu werden, und sie werden bei Bedarf *gebrüchen*, d.h. sie werden mit *brutaler Gewalt* entzogen. Das in Abschnitt 5.3 propagierte Modell ermöglicht dies in Extremfällen zwar auch, verfolgt aber im Normalfall die Idee, dass ein *Konsens* über die Verteilung von Locks hergestellt werden sollte. Opportunistische Locks stellen dagegen ein eigenständiges Konzept dar, das neben den „normalen“ Locks alternativ oder zusätzlich zur Performanz-Steigerung existiert. In dieser Hinsicht unterscheidet sich der hier vorgestellte Ansatz zu dem von Microsoft grundlegend.

<sup>23</sup>Ein Ansatz zur Vermeidung von Deadlocks in Datenbanken ist *konservativer Locking* (siehe z.B. [GR93, VGH93]), bei dem alle von einer Transaktion benutzten Ressourcen einmalig am Anfang atomar angefordert werden müssen (vgl. auch das „Handwerker-Problem“ in [Jür73], das eine Verallgemeinerung des bekannten Philosophen-Problems [Lam74] darstellt); dies setzt jedoch entweder genaue a-priori-Kenntnisse über das zukünftige Verhalten voraus, oder es schränkt die Parallelität extrem stark durch unnötige Spekulationen auf später meistens doch nicht wirklich genutzte Ressourcen ein. Bei vorher nicht bekanntem Verhalten der Transaktionen führen inkrementell nach tatsächlichem Bedarf gesetzte 2-Phasen-Sperren zu einem Deadlock-Problem, das sich *a priori* nicht vermeiden lässt (die posteriori-Erkennung ist dagegen relativ einfach; damit werden jedoch unvorhersehbare Rollbacks in Kauf genommen). Deadlocks lassen sich zwar durch halbgeordnetes (zyklenfreies) Setzen von Locks vermeiden, doch auch dafür muss man das zukünftige Verhalten der Transaktionen *a priori* kennen, was in der Praxis nur selten gegeben sein dürfte. Wenn man Rollbacks wie bei einigen Betriebssystem-Anwendungen unbedingt vermeiden muss, und/oder wenn man das zukünftige Verhalten von Konsumenten nicht kennt, dann hilft im allgemeinen nur, das Scheduling durch Anfordern eventuell später doch nicht wirklich benötigter Locks einzuschränken, d.h. den möglichen Parallelitätsgrad zu senken. Daher ist es von immenser Wichtigkeit, solche Locks zu verwenden, die sich gegenseitig möglichst wenig stören. Das *multiversion*-Modell (Kapitel 7) kann hierzu ebenfalls einen Beitrag leisten.

<sup>24</sup>Deadlocks können nicht nur bei Transaktionen, sondern auch bei `multiuser`- oder `multiversion`-Verhalten von Anwendungen auftreten, die mit der Anwesenheit paralleler Aktivitäten umgehen können (z.B. `code_reentrant` aus Abschnitt 2.6). Ich sehe Deadlocks nicht als dem Transaktions-Paradigma inhärent, sondern der Parallelverarbeitung auf gemeinsamen Daten schlechthin. Im Unterschied zum klassischen Transaktions-Paradigma, das die Auswirkungen und Folgen von Parallelverarbeitung vor den Konsumenten zu verstecken sucht, macht das `singleuser`- und `multiuser`-Modell diese Problematik *explizit*. Dies stellt zwar höhere Anforderungen an die Programmierer, ermöglicht aber feiner gesteuerte Reaktionen auf Fälle von Deadlocks, z.B. indem ein Prozess ein Signal erhält oder zur Rückgabe eines einzelnen Locks gebeten oder

sollte, wann immer es möglich ist (sofern man Wert auf Parallelität, Skalierbarkeit und Performanz legt)<sup>25</sup>. In diesem Sinne ist `adaptor_synchronize` nur als Notbehelf zu verstehen und einzusetzen.

#### 4.1.8.2 Adaption zwischen verschiedenen `transfer_size`-Werten

Im Idealfall sollten sich Baustein-Implementierungen nicht um das Problem kümmern müssen, dass die `transfer_size` eines Ausgangs mit derjenigen eines Eingangs zusammenpassen muss. Diese Umwandlungs-Aufgabe sollte an `adaptor_*` delegiert werden können.

Im Allgemeinen kann es vorkommen, dass die beteiligte Transfergrößen keine Teiler voneinander bilden. Dieser Fall taucht in der Praxis kaum auf, weil aus guten Gründen nur Zweierpotenzen<sup>26</sup> als feste Transfergrößen benutzt werden. Falls er dennoch auftreten sollte<sup>27</sup>, kann man sich auf folgende Weise behelfen: man bestimme den ggT (größter gemeinsamer Teiler) der beiden vorkommenden Transfergrößen. Dann schalte man zwei `adaptor_*`-Instanzen hintereinander, wovon die erste von der Eingangs-Transfergröße auf den ggT hinunter transformiert, die zweite vom ggT wieder auf die Ausgangsgröße hoch transformiert. Im Folgenden beschränke ich mich daher auf die Annahme, dass eine der beiden Transfergrößen ein Vielfaches der anderen darstellt. Es sind zwei Fälle zu unterscheiden:

1. Hochtransformation von einer kleinen `transfer_size` am Eingang zu einer größeren am Ausgang
2. Hinuntertransformation von einer großen `transfer_size` am Eingang zu einer kleineren am Ausgang

Bei der Hochtransformation besteht das Problem, dass der Konsument an seinem Eingang und damit auch am Ausgang des `adaptor_*` erwartet, dass eine `get`-Operation einen physisch zusammenhängenden Datenblock mit mindestens seiner Transfergröße liefert. Da am Eingang des `adaptor_*` eine kleinere Transfergröße eingestellt ist, braucht der dort angeschlossene Produzent nicht unbedingt Datenblöcke mit dieser Größe und/oder nicht unbedingt an entsprechend ausgerichteten Adressgrenzen im physischen Hauptspeicher zu liefern (allerdings wird jedem Produzenten geraten, dies dennoch zu erfüllen, wenn es im Einzelfall ohne größere Kosten möglich ist).

gezwungen wird, ohne dass deswegen gleich der Totschlag-Hammer eines Rollbacks geschwungen werden muss.

<sup>25</sup> Ich halte Transaktionen nicht deswegen für sehr nützlich, weil sie eine `singleuser`-Illusion ermöglichen, sondern weil sie den Parallelitätsgrad im Sinne einer optimistischen Strategie steigern können und die Rollback-Funktionalität explizit anbieten; diese ist auch bei `multiuser`-Verwendung von Transaktionen sehr nützlich.

<sup>26</sup> Ausnahmen mit „krummen“ Sektorgrößen kommen vor, z.B. ältere Platten-Geräte und einige moderne dazu kompatible Spezial-Festplatten von IBM, die zur Abspeicherung von ISAM-Informationen gedacht sind. Ferner gibt es in den CD-ROM-Standards einige krumme Sektorformate, auf die von aktuellen Betriebssystemen nur der rohe ungepufferte Zugriff unterstützt wird.

<sup>27</sup> Das Konzept der Nester unterstützt ausdrücklich völlig beliebige Transfergrößen. Wer allerdings Primzahlen als Transfergröße benutzt, der ist selbst schuld und darf in der Folge keine besonders gute Performance erwarten.

Im Allgemeinen ist es wünschenswert, dass der Eingang an `multiuser`- oder `multiversion`-Verhalten teilnehmen kann und damit weitere parallel arbeitende Konsumenten zulässt, die unabhängig voneinander Blöcke mittels `get` anfordern und halten können. In diesem Fall kommt ein Verschieben von Blöcken im Hauptspeicher an „günstigere“ physische Adressen im allgemeinen nicht in Frage<sup>28</sup>. Wenn man den Eingang nicht im `singleuser`-Modus betreiben möchte, dann bleibt wohl oder übel nichts anderes übrig, als *physische Kopien* von Datenblöcken herzustellen.

Eine von einem `adaptor_*` hergestellte physische Kopie muss auch von diesem Baustein bezüglich der Referenzzähler, Speicherfreigabe usw. verwaltet werden; beim letzten `put` auf einer zum Schreiben freigegebenen Kopie müssen die Änderungen auf das Original bzw. auf die kleineren Original-Teile rückübertragen werden.

Bei genauerer Betrachtung des Problems fällt auf, dass die Herstellung von physischen Kopien eventuell die Chance birgt, die mögliche Parallelität von Operationen dadurch zu erhöhen, dass die verschiedenen Versionen ausdrücklich für ein `multiversion`-Modell am Ausgang genutzt werden, auch wenn der Eingang nur `multiuser`-Verhalten weiterreicht. Eventuell bietet es sich an, die Funktionalität von `adaptor_multi` gleich hier mit zu integrieren, bzw. `adaptor_multi` gleich mit der Hochtransformations-Fähigkeit auszustatten, so dass ein einziger Baustein-Typ beide Aufgaben erledigt.

Zur Hinuntertransformation: hier besteht das Problem der Transfergrößenunterschiede nicht, da eine kleinere teilbare Transfergröße die Bedingungen der größeren bereits von selbst erfüllt. Bei der Weitergabe physischer Adressen an den Ausgang ist lediglich zu beachten, dass diese *innerhalb* eines Blocks liegen können, der vom Eingang geliefert wurde. Bei der Rückgabe mittels `put` muss eine solche physische Adresse wieder auf die `transfer_size` des Eingangs normiert werden, was durch Divisionen und Multiplikationen, bei Zweierpotenzen auch durch Ausmaskieren von Adressbits erfolgen kann.

Die Hinuntertransformation hat jedoch mit einem Problem zu kämpfen, das bei der Hochtransformation nicht existiert, weil es dort aus Symmetriegründen bereits von selbst erfüllt ist: die `move`-, `clear`- und `delete`-Operationen können vom Konsumenten am Ausgang in kleineren Portionen angefordert werden, als sie vom Produzenten am Eingang zur Verfügung gestellt werden. Daher ist eine einfache Durchreicherung dieser Operationen an den Produzenten im Allgemeinfall leider nicht möglich.

Zur Lösung dieses Problems könnte ein `adaptor_*` die `move`-Operation komplett neu implementieren, ohne davon

<sup>28</sup> Da es vermutlich doch relativ häufig vorkommt, dass im Einzelfall entweder nur ein einziger Konsument gerade Datenblöcke hält, oder dass eine Rückforderung mittels der später besprochenen `notify_*`-Operationen möglich ist, dürften Verschiebungen in vielen Fällen lohnend sein. Dazu müssen aber die Operationen auf Nester in irgend einer Weise um einen Verschiebemechanismus auf physischer Adressebene erweitert werden, was ich hier nicht im Detail durchführen möchte. Eine Möglichkeit dazu wäre eine Verschiebeoperation `try_move`, mit der der Konsument aufgefordert werden kann, eine Verschiebung durchzuführen, sofern sie ohne Nebeneffekte auf andere Benutzer möglich ist. Alternativ dazu könnte die `get`-Operation um Parameter erweitert werden, die einen Allokationswunsch ausdrücken, dem der Produzent aber nicht zwingend nachkommen muss.

Gebrauch zu machen, dass der Eingang bereits ein dynamisches Nest bereitstellt. Dies widerspricht jedoch dem Ziel, die Bausteine zur „möglichst orthogonalen“ Zerlegung der in Betriebssystemen vorkommenden Aufgaben einzusetzen. Was bedeutet jedoch „möglichst orthogonal“?

Für die Beurteilung der Orthogonalität und Homogenität eines Baustein-Verhaltens sehe ich als wichtiges Kriterium, ob eine Transformation wieder durch eine rückläufige Transformation aufgehoben werden kann, oder zumindest bis auf einen konstanten Rest aufgehoben werden kann (Prinzip der Kompensierbarkeit). Dies würde bedeuten, dass eine Hinuntertransformation mit nachgeschalteter Hochtransformation auf die ursprüngliche `transfer_size` nichts am Inhalt eines Nestes ändern sollte, also insgesamt eine idempotente Abbildung darstellen sollte.

Leider ist dies aus informationstheoretischen Gründen in vollkommener *Reinform* nicht möglich: Um die Aufteilung einer gegebenen festen Anzahl von Nutzbytes auf einen ebenfalls festen größeren Adressraum mit Löchern zu verwalten, muss irgendwo Platz für die Darstellung dieser Zustandsinformation aufgewandt werden. Wenn man den Nutzen auf den größeren Adressraum irgendwie (z.B. zufälliger Art Schweizerkäse) verteilt, dann gibt es für die Anzahl solcher möglichen Verteilungen um so mehr Möglichkeiten, je kleiner die `transfer_size` gewählt wird (der Schweizerkäse darf bildlich gesprochen immer kleinere Löcher und auch mehr Löcher enthalten, obwohl er weder sein Gewicht noch sein Volumen ändert). Welche dieser vermehrten Möglichkeiten konkret vorliegt, muss irgendwo gemerkt und abgespeichert werden, was aus informationstheoretischen Gründen einen Platzbedarf zur Folge hat, der mit immer kleinerem `transfer_size` zumindest im *Worst Case* ansteigt<sup>29</sup>. Hieraus folgt, dass es im Allgemeinen nicht möglich ist, die `transfer_size` nach unten zu transformieren, ohne dass irgendwo zusätzlicher Platz für die Speicherung der nunmehr feineren Abbildungsmöglichkeiten für den Definitionsbereich des Nestes verbraucht wird.

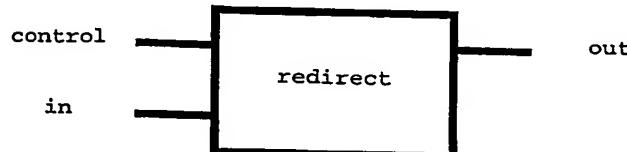
Dies bedeutet jedoch nicht, dass zusätzlicher Platz in allen verbraucht werden muss. Ich kann von einer Hinuntertransformation folgende wünschenswerte Eigenschaft annehmen:

Falls am Ausgang der Hinuntertransformation nur solche Operationen ausgeführt werden, die auch ohne die Hinuntertransformation stattfinden könnten, weil sie (zufälligerweise) die Bedingungen der größeren `transfer_size` am Eingang bereits erfüllen, dann sollte am Ausgang der Hinuntertransformation derselbe Nest-Zustand sichtbar sein, als wäre die Hinuntertransformation nicht vorhanden. Dies bedeutet u.a., dass in diesem Fall kein Platz für interne Verwaltungsinformationen abgezwackt werden darf (der Sachverhalt als solcher kann im Meta-Nest mit geringen Platzkosten vermerkt werden). Eine derartige Hinuntertransformation stellt in gewissem Sinn eine *Verfeinerung*

des Eingangs-Nestes am Ausgang zur Verfügung.

Zu realisieren ist eine solche Verfeinerung im Prinzip dadurch, dass nur die Unterschiede zu der größeren Eingangstruktur intern verwaltet werden. Bei dieser Verwaltung tritt u.a. das in der Literatur bekannte Problem des internen Verschnitts auf, für das es mehrere Lösungsansätze gibt<sup>30</sup>. Auf weitere Details möchte ich in dieser Arbeit nicht eingehen.

#### 4.1.9 redirect



Dieser Baustein realisiert die Funktionalität von Hard- und Softlinks<sup>31</sup>, sowie von Mount-Punkten und -Tabellen. Er hat zwei Eingänge und einen Ausgang. Vom ersten Eingang wird standardmäßig das Meta-Nest (falls dieses leer sein sollte, auch das Daten-Nest) abgefragt, ob es einen Pfad im Auto-Instantierungs-Netzwerk enthält. Der zweite Eingang wird daraufhin mit dem angegebenen (ggf. noch zu konstruierenden) Ausgang automatisch verbunden, der dann wiederum am Ausgang unverändert zur Verfügung steht. Weitere Details sind in Abschnitt 4.2 erklärt.

#### 4.1.10 cow



Dieser Baustein realisiert die von konventionellen privaten Mappings bekannte Copy-on-Write-Strategie. Er hat zwei Eingänge, die mit `orig` und `tmp` bezeichnet sind, sowie einen mit `merged` bezeichneten Ausgang.

Im initialen Zustand ist das mit `tmp` bezeichnete Eingangs-Nest leer, und am mit `merged` bezeichneten Ausgang erscheint exakt dieselbe Nest-Zustand wie am `orig`-Eingang. Die Aufgabe besteht in der *Isolation* des `orig`-Eingangs von allen Änderungen, die vom Konsumenten hinten am `merged`-Ausgang in Auftrag gegeben werden. Jegliche Änderungen am Nest-Inhalt des Ausgangs oder an seinem Adressraum müssen ausschließlich im `tmp`-Nest eingetragen und zwischen-gepuffert werden, damit sie keine tatsächlichen Änderungen am `orig`-Eingang bewirken.

<sup>29</sup>Wovon der wichtigste meiner Ansicht nach die *weitgehende Vermeidung* von Verschnitt darstellt.

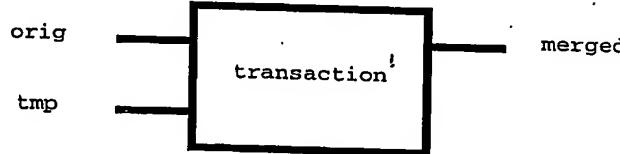
<sup>30</sup>Zur Nachbildung eines  $n$ -fachen Hardlinks muss dieser Baustein  $n$ -fach instantiiert werden, außerdem muss noch eine Verwaltungslogik für die Freigabe hinzukommen, die beispielsweise von einem gesonderten Verwaltungsbaustein erledigt werden kann, der sozusagen die „Aufsicht“ über alle Namensänderungen und Verschiebungen aller beteiligten Namen übernimmt. Alternativ dazu kann dieser Baustein auch so ausgeführt werden, dass er  $n$  Steuer-Eingänge hinzubekommt, die zusammen diese Aufgabe übernehmen.

Die Implementierung von write-Transferoperationen bzw. get im Schreibmodus ist relativ einfach und folgt der bekannten Methodik, wobei Löcher im Definitionsbereich von tmp direkt ausnutzbar sind. Schwieriger ist die Bereitstellung von move-Operationen. Eine Analyse dieses Teilproblems ergibt, dass es starke Verwandtschaft mit der Adressverschiebungs-Problematik bei den map-Bausteinen besitzt. Eine mögliche Realisierung ist daher die interne Verwendung eines map-Bausteins, der allerdings auch mit Löchern im orig-Nest korrekt umgehen können sollte.

#### 4.1.11 transaction

Im Unterschied zu Datenbanken, wo Transaktions-Identifier (TIDs) meist fest mit Prozessen verknüpft sind, verstehe ich unter einer Transaktion eine *logische Sicht* auf einen Datenbestand, die die bekannte ACID-Eigenschaft (oder Eigenschaften anderer Transaktions-Paradigmen) besitzt, und die von beliebig vielen Prozessen kooperativ (bzw. bei Verwendung zusätzlicher Sperren auch im multiuser- oder multiversion-Modell) nutzbar ist. Durch die Aufgabe fester Zuordnungen zwischen Prozessen und Datenräumen wird insbesondere auch ermöglicht, dass ein Prozess an mehreren Transaktionen, auch parallel, teilnehmen darf. Das Standard-Szenario einer festen 1:1 Zuordnung zwischen Prozessen und Datenräumen ist als Spezialfall in diesem Modell enthalten.

##### 4.1.11.1 Sequentielle Transaktionen



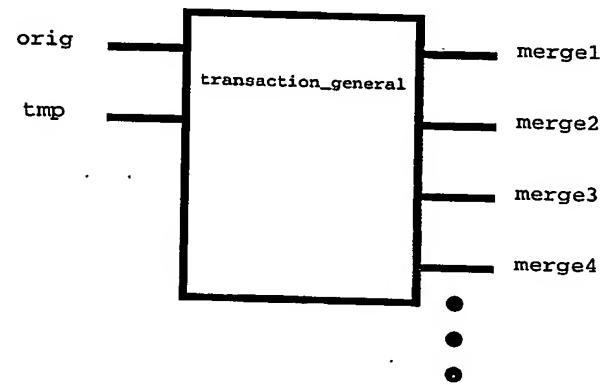
Ein cow-Baustein realisiert bereits einen wichtigen Teil der bekannten *Isolations-Funktionalität* von Transaktionen: um eine Rollback-Operation zu simulieren, braucht man lediglich den Inhalt des tmp-Nestes zu vergessen und den merged-Ausgang wieder auf den initialen Zustand des orig-Eingangs zu setzen. Ein transaction-Baustein lässt sich im Prinzip als leicht modifizierter cow-Baustein realisieren, der seine grundsätzliche Funktionsweise von cow erbt.

Die Commit-Operation lässt sich prinzipiell dadurch realisieren, dass man einfach *gar nichts* tut: man hört auf, Änderungen am merged-Ausgang vorzunehmen, und betrachtet seinen Zustand als „eingefroren“. Ein derartig „eingefrorenes“ Nest dient nun insbesondere als Ausgangspunkt für eine zeitlich nachfolgende (nicht-parallele) Transaktion, deren orig-Eingang mit dem merged-Ausgang der jeweils letzten erfolgreich abgeschlossenen Transaktion verbunden wird, so dass im Endeffekt lange Ketten entstehen, die alle historischen Zwischenschritte der von den Transaktions-Operationen ausgelösten Zustände des Nestes repräsentieren. Der Ausgang der letzten erfolgreich abgeschlossenen transaction-Instanz einer Kette wird *Aktualversion* genannt. Hinter der Aktualversion darf vorläufig nur eine einzige, noch nicht abgeschlossene Transaktion angeschlossen werden; ansonsten könnten divergierende

Versions-Splits<sup>32</sup> entstehen, die dem Konzept einer einheitlichen logischen Sicht widersprechen.

Es ist klar, dass solche Ketten nicht beliebig lang werden dürfen. Die Entfernung von Bausteinen ohne Änderung der Semantik ist auf folgende Weise möglich: eine abgeschlossene Transaktion darf ohne weiteres *invariante interne Operationen* ausführen, die von außen durch die (gegenüber cow neu hinzukommende) Operation *integrale* angestoßen werden und bewirken, dass das Ausgangs-Nest weiterhin von außen betrachtet denselben eingefrorenen Zustand behält, während der Inhalt des tmp-Eingangs in das orig-Eingangsnest *integriert* wird, so dass das tmp-Nest immer kleiner wird, bis es schließlich völlig leer wird. Bei dieser Integrations-Operation wird der Inhalt des orig-Eingangs so verändert, dass er schließlich mit dem eingefrorenen merged-Ausgang übereinstimmt; damit ist die Isolation aufgehoben. Ab diesem Zeitpunkt darf man den nunmehr völlig statuslos gewordenen transaction-Baustein aus der Kette entfernen, ohne dass sich Seitenefekte ergeben.

##### 4.1.11.2 Parallele Transaktionen



Mit der soeben vorgestellten Methodik lassen sich nur rein sequentiell nacheinander ablaufende Transaktionen modellieren.

Zur Herstellung echter Parallelität zwischen mehreren transaction-Instanzen müssen diese unbedingt an denselben Aktualitätsversion angeschlossen werden (sofern sie nicht absichtlich mit einem veralteten Zustand beginnen wollen). Wenn man dies mit getrennten Einzel-Bausteinen für jede Transaktion realisieren würde, dann müssten sich die verschiedenen Transaktions-Sichten über diesen oder einen anderen Anschluss-Punkt auf dem aktuellen Stand halten und jede für sich den aktuellen Status an ihrem Eingang nachvollziehen, was im Prinzip möglich wäre, aber eine Vervielfachung von immer gleichartigen Operationen an mehrere Stellen bewirken würde<sup>33</sup>. Um diesen Aufwand zu sparen, ist es günstiger, eine konventionelle Implementierung von Transaktionen (siehe z.B. [GR93]), eventuell auch von verschiedenen Transaktions-Modellen (siehe z.B.

<sup>32</sup>Hierfür gibt es reizvolle Anwendungen, sofern Methoden zur Reintegration von Splits vorhanden sind. Beispiele für Splits treten u.a. bei Versions-Verwaltungssystemen für Software-Quelltexte auf.

<sup>33</sup>Für besondere Anwendungen, beispielsweise extrem hoher Ausfallsicherheit, könnte diese Idee eventuell trotz ihrer hohen Kosten interessant sein.

- [Elmbe]) in einen Baustein zu verpacken, der einen internen zentralen Puffer für die Verwaltung aller vorkommenden Versionen benutzt, und/oder dessen Eingang im multi-version-Modus arbeitet (dann muss allerdings auch die restliche Infrastruktur des Systems mitspielen).

Bei der Kombination von Transaktionen mit Schutzbereichen kann das Baustein-Konzept mit Funktionalitäten aufwarten, die in konventionellen Betriebssystem-Implementierungen einen *erheblichen* Aufwand verursachen würden. Ein Beispiel ist das Exception-Handling in konventionellen Laufzeitumgebungen. Dieses kann auf einen Fehler nur noch *reagieren*, ihn aber meist nicht *reparieren*, da es beim Auftreten des Fehlers bereits „zu spät“ ist. Transaktionen bieten die Möglichkeit, auf frühere Zustände von Adressräumen zurückzusetzen und erneute Versuche zu starten, und zwar auch auf ganze Kollektionen und Konfigurationen von Adressräumen (*spekulative Ausführung*).

Sehr interessant dürfte auch die Kombination von Transaktions-Bausteinen mit den nun folgenden Baustein-~~sein~~en, die verteilte Systeme in Netzwerken ermöglichen und zwar je nach Einsatz-Stelle auf verschiedenen Ebenen einer Baustein-Hierarchie.

#### 4.1.12 remote

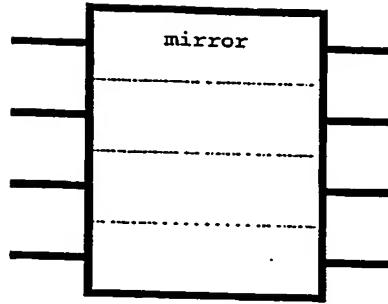


Dieser Baustein implementiert das Client-Server-Paradigma. Ein Nest wird auf einem anderen Rechner so verfügbar gemacht, dass seine Eigenschaften nicht von einem lokalen Nest unterscheidbar sind.

Realisierung des Netzwerk-Protokolls kann statuslos erfolgen, wenn man eine *buffer*-Instanz nachschaltet, die durch ihr Caching den Datenverkehr über die interne Netzwerk-Verbindung in vielen Fällen reduziert und nebenbei die Latenzzeit vieler Operationen senkt. Bei einer statuslosen Realisierung hält der Client-Teil keinerlei Status-Informationen über den Zustand des Nestes vor, sondern muss jede einzelne Operation an den Server durchreichen. Dies führt zu einer hohen Einfachheit, Robustheit, Unabhängigkeit und Absturzsicherheit.

Fragen der Einbruchssicherheit in Netzwerke, Verschlüsselung, Authentifizierung usw. sind eine interne Angelegenheit der Baustein-Implementierung; in dieser Arbeit wird hierauf nicht weiter eingegangen.

#### 4.1.13 mirror



Dieser Baustein realisiert das Konzept von Verteiltem Gemeinsamen Speicher (distributed shared memory).

Eine Baustein-Instanz von *mirror* darf sich über mehrere physikalisch getrennte Rechner erstrecken, die miteinander über interne (nicht von außen sichtbare) Kommunikationsmechanismen in Verbindung stehen (hierfür bietet sich insbesondere Gruppenkommunikation an). Die Verteilung der Baustein-Instanz über mehrere Rechner wird im Bild durch gestrichelte Linien angedeutet.

Ein- und Ausgänge sind im Grundmodell paarweise in der gleichen Anzahl vorhanden. Die Ausgänge stellen dieselbe Funktionalität dar, wie sie in einem nicht-verteilten System von der gemeinsamen Verdrahtung mehrerer Eingänge auf denselben Ausgang erfüllt wird: überall herrscht die gleiche logische Sicht, das Nest ist logisch betrachtet identisch und ermöglicht Kooperation im *multiuser*- oder *multiversion*-Modell.

Diese knappe, aber vollständige Funktionsbeschreibung lässt viele Möglichkeiten für die Realisierung zu, die im Wesentlichen den bekannten Techniken aus der Literatur folgen kann (siehe z.B. [TSF90, Doe96, Esk96]). Was die Eingänge enthalten, kann bei verschiedenen *mirror\_\**-Typen stark voneinander abweichen. Ich werde zwei Extremfälle kurz erläutern:

Bei einer Realisierung namens *mirror\_replicate* enthalten alle Eingänge den gleichen Nest-Zustand wie die Ausgänge. Jede an irgendeinem Ausgang ankommende Änderung wird sofort auf alle Eingänge aller verteilten Teil-Instanzen durchgereicht. Bei parallelem Zugriff auf mehreren Rechnern erfordert dies im Regelfall verteilte interne Synchronisationsoperationen, um die *Konsistenz* jederzeit zu wahren.

Das dadurch verursachte Problem relativ hoher Latenzzeiten lässt sich durch rechner-lokales Nachschalten je einer *buffer*-Instanz hinter die Ausgänge in vielen Fällen abmildern; daher kann die interne Realisierung der Kommunikationsprotokolle weitgehend statuslos erfolgen.

Eine andere Realisierung namens *mirror\_primarycopy* besitzt nur einen einzigen Eingang, der einem ausgezeichneten Rechner fest zugeordnet ist und stets dort verbleibt, auch wenn weitere Ausgänge auf anderen Rechnern dynamisch hinzugefügt oder weggenommen werden.

Schließlich möchte ich noch auf Mischformen zwischen den beiden Extremen der vollständigen Replikation aller Daten und der replikationslosen Primärkopie hinweisen. Dazu gehören Baustein-Varianten, die ihre Eingänge nicht für Replikate, sondern für die Bereitstellung der *Summe* aller vorhandenen Platzes zu verwenden (disjunkt verteilter

Speicher). Es ist prinzipiell möglich, einen Baustein `mirror_general` zu schaffen, der sowohl die Extremfälle der vollständigen Replikation als auch der Primärkopie, als auch beliebige Mischformen mit teilweiser Replikation, als auch Mischformen mit disjunkt verteilem Speicher abdeckt.

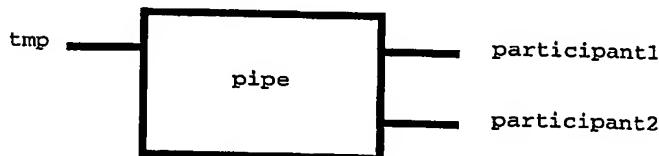
Zu erwähnen ist weiterhin, dass die RAID-Konzepte (Redundant Array of Inexpensive Disks, vgl. [CLG+93]) als Spezialfall von `mirror` mit einer je nach RAID-Level besonderen Art der Redundanz-Verteilung aufgefasst werden können: es wird lediglich auf eine Verteilung der Ein- und Ausgänge auf verschiedene Rechner verzichtet. Eine effiziente Realisierung von `mirror` sollte sich daher darum bemühen, dass lokale Replikation keinen merklichen Overhead zur Folge hat. Damit würde es sich auch für die Aufgabengebiete eignen, in denen bisher Software-RAID eingesetzt wird.

Implementierungen von `mirror` sollten auf jeden Fall den Grundideen der Ausfallsicherheit hohe Aufmerksamkeit schenken und z.B. mit dem Fall einer Netzwerk-Partitionierung (vgl. [DGMS85]) sinnvoll umgehen können. Eine detaillierte Behandlung dieser Konzepte würde den Rahmen dieser Arbeit sprengen. Es wäre interessant zu ergründen, inwieweit das Baustein-Konzept und einige der hier vorstellten Bausteine bei der internen Realisierung von `mirror` Vorteile bringen können.

Schließlich ist zu erwähnen, dass sich die Funktionalität von Prozess-Migration (vgl. [Smi88]) als Spezialfall von verteilt ablaufenden Kontrollflüssen auf verteilten gemeinsamen Schutzbereichen bzw. durch Wechseln von Kontrollflüssen auf andere Rechner unter Benutzung gespiegelter Schutzbereiche darstellen lässt. Hierzu ist erforderlich, dass die verwendeten `mmu_*`-Bausteine `multiuser`-Verhalten implementieren<sup>34</sup>.

Die Verteilung von Nestern ist auch zur Herstellung verteilter gemeinsamer Namensräume sehr nützlich; `mirror` ist somit die wesentliche Grundlage für ein verteiltes Betriebssystem.

#### 4.1.14 pipe



Zum Verständnis des `pipe`-Entwurfs muss man sich nochmals in Erinnerung rufen, dass eine Leitung eine Hierarchie-Beziehung zwischen Baustein-Instanzen darstellt und nichts direkt mit Datenfluss-Richtungen zu tun hat. In abstrakter Sicht sind daher die Teilnehmer an einer Pipe prinzipiell gleichberechtigt, sie werden daher an verschiedenen Ausgängen angeschlossen. Der interne Status einer `pipe`-Instanz wird in einem `tmp`-Eingang gehalten, an den sich nicht nur `device_ramdisk` anschließen lässt,

<sup>34</sup>Welche Auswirkungen dies auf die Performanz haben kann, ist eine andere Frage. Im Prinzip haben wir damit eine NUMA-Architektur in Software realisiert.

sondern z.B. auch Puffer-Nester, die nicht in den Hauptspeicher eines Kleinrechners passen würden.

## 4.2 Instantiierung von Bausteinen

Zur Instantiierung der Bausteine wird irgendein Mechanismus benötigt. Dieser sollte außerhalb oder oberhalb der eigentlichen Baustein-Hierarchie liegen, da es sich um einen übergeordneten Kontroll-Mechanismus handelt. Nach dem Grundsatz der Trennung in Mechanismen und Strategien sind folgende Teilbereiche zu verwenden:

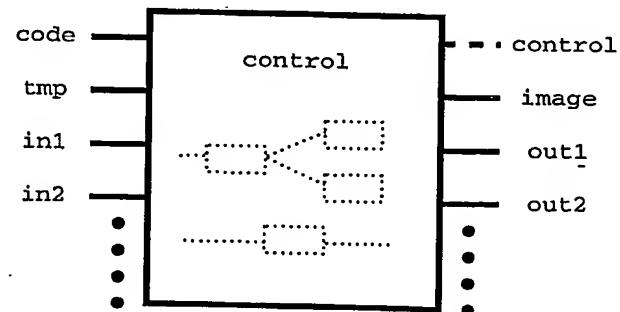
1. Der eigentliche Instantiierungs-Mechanismus (technische Durchführung der Instantiierung)
2. Die logische Kontrolle, welche Instantiierung zu welchem Zeitpunkt und aus welchem Anlass ausgeführt werden soll

Diese Trennung hat den Vorteil, dass insbesondere für Punkt 2 mehrere verschiedene Verfahren gleichzeitig oder konkurrierend eingesetzt werden können.

### 4.2.1 Der Mechanismus

Für die technische Durchführung der Instantiierung schlage ich vor, einen Baustein namens `control` einzusetzen, von dem es auf jedem Rechner beliebig viele Instanzen geben kann, wobei im Normalfall jedoch eine Instanz ausreicht, die ihrerseits beim Urstart auf eine Weise instantiiert worden sein muss, die außerhalb des normalen Instantiierungs-Mechanismus liegt (und logischerweise auch liegen muss<sup>35</sup>).

Der `control`-Baustein verwaltet alle von ihm erzeugten Baustein-Instanzen samt Verdrahtung. Er selbst benötigt ebenfalls Betriebsmittel, beispielsweise den Maschinencode de von sich selbst.



Ich schlage vor, diese beiden Funktionsbereiche logisch zu trennen: Die von einer `control`-Instanz verwalteten Baustein-Instanzen können i.a. in einem anderen (virtuellen) Adressraum (der wiederum in Schutzbereiche aufgeteilt sein kann) liegen als die `control`-Instanz selbst; damit sind z.B. beliebige hierarchische Schachtelungen

<sup>35</sup>Nach dem Urstart müssen bereits Bausteine vorhanden sein, damit das System in Betrieb gehen kann (unverzichtbare Gerätetreiber und Äquivalent eines Root-Dateisystems).

von Adressräumen oder ganzen virtuellen Maschinen möglich<sup>36</sup>. Die steuernde control-Instanz bedient sich des tmp-Eingangs, um darin den gesamten notwendigen Status zu halten, und stellt am image-Ausgang ein „Prozessabbild“ bzw. eine Menge von Schutzbereichen zur Verfügung, die in einer nachfolgenden (ggf. über Zwischenbausteine wie union angeschlossene) mmu\_i386 oder mmu\_dummy-Instanz zur eigentlichen Ausführung gebracht werden. Damit ist eine Separation zwischen dem Adressraum möglich, der die control-Instanz beherbergt, und dem Adressraum, in dem die verwalteten Instanzen laufen sollen. Durch diese Trennung ist es u.a. möglich, Teile eines Betriebssystems in „Benutzer“-Adressräumen ablaufen zu lassen. Benutzer können auf diese Weise von einander unabhängige Subsysteme schaffen, die von der Funktionalität her virtuellen Betriebssystemen gleich kommen. Damit verschwimmt die klassische Einteilung in Kern- und Benutzer-Zuständigkeiten; die Verteilung dieser Zuständigkeiten ist nur noch eine Frage der Konfiguration.

Eine Trennung zwischen der control-Instanz und ihrem image in separate Adressräume ist jedoch keine <sup>37</sup>: wenn man den image-Ausgang in die gleiche \*-Instanz einblendet, in der auch der control-Baustein beheimatet ist, kann man ohne weiteres Single-Address-Space-Strategien<sup>38</sup> fahren. Als weiterer Sonderfall ist es prinzipiell auch möglich, dass ein control-Baustein sich selbst verwaltet<sup>39</sup>; in diesem Fall besteht kein Unterschied zwischen „innen“ und „außen“; eine Destruktion der control-Instanz würde dann nicht nur zur Destruktion aller verwalteten Instanzen führen, sondern nie mehr eine erneute Re-Konstruktion ermöglichen (dies kann ausnahms-

<sup>36</sup>Diese Trennung bewirkt u.a., dass sich die Abstraktionen Nest und Baustein in uniformer Weise auch in konventionellen „Benutzerprozessen“ verwenden lassen, eventuell sogar in sogenannten „Anwendungsprogrammen“ – es besteht überhaupt kein prinzipieller Unterschied zwischen „Kern“- und „Benutzer“-Adressräumen bzw. -Schutzbereichen.

<sup>37</sup>Im Extremfall kann das gesamte Betriebssystem in einem einzigen Adressraum ablaufen, was z.B. bei Echtzeit-Steuerungen Vorteile bringt. Falls Schutzbereiche genutzt werden, lässt sich trotzdem ein brauchbarer Zugriffsschutz realisieren (vgl. [C+94]). Im Unterschied zu bekannten Architekturen (vgl. [K+97]) erlaubt die hier vorgestellte Möglichkeit beliebige Zwischenstufen zwischen reinen Single-Space-Modellen und vollkommener Aufplitterung aller Funktionen getrennte mmu\_\*-Instanzen.

<sup>38</sup>Die Benutzung eines einzigen virtuellen Adressraums im gesamten System lässt sich dadurch modellieren, dass nur eine einzige mmu\_\*-Instanz vorhanden ist, in der das gesamte System abläuft. Obwohl diese Strategie in letzter Zeit hohe Aufmerksamkeit in der Literatur erhalten hat [C+94, Ros94, Ass96], sehe ich darin keinen grundlegenden Vorteil. Die Umschaltung einer Standard-MMU auf verschiedene Schutzbereiche kostet (mit wenigen Ausnahmen) im Worst Case größenordnungsmäßig ebenso viel wie die Umschaltung zwischen verschiedenen Adressräumen; lediglich der Best Case geht u.U. effizienter. In welcher Größenordnung sich dieser mögliche Performanz-Gewinn auf das Gesamtsystem auswirkt, und ob Effekte gleicher Größenordnung nicht auch durch andere architekturelle Maßnahmen erzielbar sind, sind genauer zu untersuchende Fragen. Meiner Ansicht nach stellt die Einengung auf ein festes Single-Space-Modell eine Einschränkung dar, die mit der hier vorgestellten Architektur aufgebrochen werden kann.

<sup>39</sup>Die einfachste Lösung hierfür besteht darin, dass der tmp-Eingang bereits bei der „Ur-Instantiierung“ das richtige Prozessabbild mit der Baustein-Konfiguration enthält, mit dem zu starten ist. Wenn dieses auch die steuernde control-Instanz enthält, entsteht automatisch die Selbstverwaltung. Bei der Systemgenerierung eines bootfähigen Systems wird auf einem (fremden) Rechner ein entsprechendes Prozessabbild mit allen zum Ustart notwendigen Instanzen erstellt; dieses kann dann mit konventionellen Ladern analog wie klassische Unix-Kerne geladen werden.

weise beim Herunterfahren des Rechners auch erwünscht sein).

Die Befehle zum Instantiieren / De-Instantiieren der verwalteten Instanzen sowie zum Verdrahten werden über den control-Ausgang gegeben. Eine Möglichkeit ist die Einführung weiterer Elementar-Operationen, oder die Benutzung der Schnittstelle von generischen Operationen (Kapitel 6). Da auf dieser Leitung nur Steuer-Operationen benutzt werden, ist sie gestrichelt gezeichnet. Über den control-Ausgang lassen sich fernerhin die Attribute der instantiierten Bausteine, Ein- und Ausgänge und die Verdrahtungsstruktur abfragen. Auftraggeber für diese Operationen können beliebige Bausteine sein; i.A. dürfen diese auch in image liegen<sup>40</sup>. Zu Zwecken der Kommunikation nach außen existiert eine dynamische Anzahl von Ein- und Ausgängen in\* und out\*<sup>41</sup>, die als Schnittstellen für LRPC dienen und die zu beliebigen Zwecken einsetzbar sind<sup>42</sup>.

Es ist sinnvoll, das Instantiieren / De-Instantiieren vom Konstruieren / Destruieren zu trennen: Beim Instantiieren wird lediglich Platz für die Baustein-Infrastruktur (z.B. statische Attribute) angelegt und die Verdrahtung ermöglicht. Die Parametrierung der Baustein-Instanz (z.B. Festlegen dynamischer Attribute) erfolgt spätestens beim Konstruieren. Wenn man einen Baustein nur destruiert, aber nicht deinstantiiert, wird er im Endeffekt lediglich in den Zustand der Statuslosigkeit geschaltet (d.h. nach dem Abarbeiten aller evtl. noch vorliegenden Aufträge wird der evtl. noch vorhandene interne Status auf die Eingänge abgewälzt, danach werden keine Operationen mehr bearbeitet), und er kann anschliessend durch Konstruieren wieder in Betrieb genommen werden.

Details zur internen Realisierung von control werden in dieser Arbeit nicht behandelt; die grundlegende Funktionsweise soll hier nur skizziert werden:

Im code-Eingang wird dynamisch linkbarer Maschinencode für alle instantiierbaren Baustein-Typen bereitgehalten<sup>43</sup>. Bei der erstmaligen Verwendung wird er nach tmp kopiert und dabei gelinkt. Die Instanzen der verwalteten Bausteine werden in zwei Phasen erzeugt: aus den statischen Attributen des zu instantiierten Baustein-Typs wird der Speicherplatzbedarf für eine Instanz ausgelesen und

<sup>40</sup>Hierbei kann es i.A. zu Problemen mit Rekursion kommen, wenn derartige Auftraggeber sich selbst betreffende Operationen in Auftrag geben.

<sup>41</sup>Alternativ zu diesen Ein- und Ausgängen könnte man auch LRPC-Varianten von remote benutzen; damit müssten keine Verdrahtungsleitungen zwischen Kommunikationskanälen auf demselben Rechner verwaltet werden. Mir scheint jedoch die explizite Verwaltung von Leitungen durch eine übergeordnete Instanz sicherer zu sein, da eine Leitung bereits eine rudimentäre Form von Authentifizierung bereit stellt, die bei geeigneter Abschottung der Subsysteme (z.B. Partitionierung in Schutzbereiche) von einem vertrauenswürdigen „logischen Supervisor“ einbruchssicher gemacht werden kann, ohne zu aufwenigeren Massnahmen wie Verschlüsselung beim LRPC greifen zu müssen.

<sup>42</sup>Dies ist nicht nur zur Herstellung mehrerer virtueller Rechnerpartionen interessant oder zu deren Kommunikation über gemeinsame Daten wie „Dateisysteme“, sondern beispielsweise auch zum Checkpoint / Restart einer einzigen Betriebssystem-Instanz, wenn man beispielsweise vor tmp eine transaction-Instanz vorschaltet. Eine weitere Anwendung wäre beispielsweise das Vorhalten eines kompletten Ersatz-Betriebssystem-Abbildes, das bei Störungen einspringen kann.

<sup>43</sup>Um reibungslose Re-Instantierungen bzw. Re-Konstruktionen zu garantieren, darf der Inhalt von code zur Laufzeit nur um neue Baustein-Typen bzw. um neue (fehlerbereinigte) Versionen erweitert werden, nicht jedoch abgeändert werden.

entsprechender Platz in `tmp` reserviert. Anschließend wird eine Instantiierungs-Routine aufgerufen; diese kann bei Bedarf weitere Aktionen ausführen<sup>44</sup>. Nach der Instantiierung sollte im Regelfall die Verdrahtung stattfinden; hierbei kann bereits auf Konflikte zwischen statischen Kompetenz-Attributen und Verhaltens-Attributen geprüft werden. Die Konstruktion erfolgt ebenfalls durch Aufruf einer baustein-spezifischen Routine mit passenden Parametern. Da u.U. erst zu diesem Zeitpunkt dynamische Attribute festgelegt werden, kann die Konstruktion auch fehlschlagen. Destruktion und De-Instantiierung funktionieren analog.

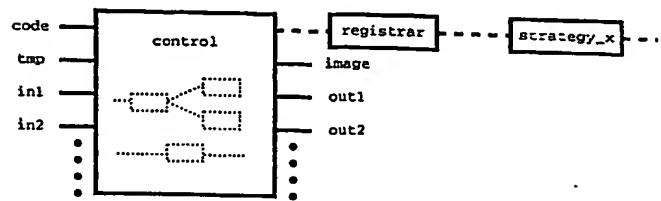
#### 4.2.2 Einige mögliche grundlegende Strategien

Bei den *Anlässen* für eine Instantiierung sind mehrere mögliche Szenarien zu unterscheiden:

1. Manche Baustein-Arten wie z.B. `dir_*` können nur an bestimmten vorgegebenen oder vorgebbaren Stellen einer Baustein-Hierarchie instantiiert werden, weil sie einen *Interpreter* für ein bestimmtes *konkretes Datenformat* darstellen. Es macht i.a. keinen Sinn, andere Baustein-Arten als die für die jeweilige Interpretation geeigneten zu instantiieren; häufig existiert für ein bestimmtes Datenformat nur eine einzige Interpreter-Baustein-Art (in einem gewachsenen System allerdings oft in verschiedenen *Versionen*, von denen bei der Instantiierung die geeignete ausgewählt werden muss).
2. Andere Baustein-Arten wie z.B. `mmu_*` oder geschachtelte `control`-Instanzen werden fast ausschließlich auf Veranlassung von Benutzer-Aktivitäten instantiiert.
3. Es gibt Mischformen zwischen beiden Extremen: ein Verzeichnis-Pfadname wird zwar vom Benutzer vorgegeben, die Art des jeweils zu instantiierenden Bausteins hängt jedoch vom Datenformat ab.
4. In manchen Fällen können Instantiierungen oder Re-Instantiierungen auch ohne *konkreten* äußeren Anlass geschehen, z.B. zeitgesteuert in selbsttätig optimierenden Betriebssystemen (vgl. [BR76]) oder zur Erzielung von Fehlertoleranz und Ausfallsicherheit.

Der Fall 1 lässt sich entweder direkt in `control` abhandeln, oder durch einen eigenen Baustein-Typ `registrar`, der als *Registrar* für verschiedene Datenformate dient. Datenformate sollten vorzugsweise im zugehörigen Meta-Nest beschrieben werden; es gibt aber auch auch Sonderfälle wie Platten-Partitionen oder `dir_*`-Bausteine für konventionelle Archiv-Datei Formate (wie `.tar` oder `.zip`), bei denen die Format-Erkennung durch Inspektion des Inhaltes des betreffenden Nestes erfolgen muss. Ein Registrar

verwaltet daher auch die Methoden für die automatische Format-Erkennung.



Um eine möglichst hohe Flexibilität bei verschiedenen Auslösern von Instantiierungen zu erreichen, schlage ich die Einrichtung von `strategy_*`-Bausteinen vor. Diese steuern den `control`- oder `registrar`-Baustein und erhalten von dort Informationen über vorhandene Datenformate und Kompetenzen; sie werden ihrerseits von anderen Steuer-Instanzen wie z.B. Benutzer-Operationen oder anderen `strategy_*`-Instanzen gesteuert. Im Bild sind die Steuerleitungen gestrichelt gezeichnet.

Ein Beispiel wäre ein Baustein `strategy_asciopath`, der Pfadnamen in ASCII-Codierung akzeptiert und indirekt über `registrar` bzw. `control` auf einen Verzeichnisbaum von `dir_*`-Instanzen abbildet. Ein weiterer Baustein `strategy_utf_ascii` liesse sich zwischenschalten, um Pfadkomponenten in Unicode- oder UTF-8-Codierung nach ASCII zu übersetzen und dabei UTF-8-Sonderzeichen in eine umkehrbar eindeutige äquivalente Mehrzeichen-ASCII-Repräsentation zu übertragen. Ein weiteres Beispiel wäre `strategy_ascii_bin`, der binär kodierte Zahlen-Schlüssel (z.B. Feldschlüssel in Datenbanken) aus äquivalenten ASCII-Repräsentationen erzeugt. Weitere `strategy_*`-Arten könnten z.B. die Verwaltung von URLs übernehmen; der Erweiterbarkeit und Flexibilität werden durch diese Architektur kaum Grenzen gesetzt. Um ständig wiederholende Instantiierungen / De-Instantiierungen von Bausteinen zu verhindern, kann `strategy_cache` einen „Vorrat“ von häufig benutzten Instanzen anlegen, die bei Nichtbenutzen evtl. lediglich auf „statuslos“ geschaltet werden. Die Funktionalität des klassischen `fork`-Systemaufrufs von Unix lässt sich ebenfalls durch `strategy_*`-Bausteine, diejenige von /proc-Dateisystemen durch spezialisierte Arten von Registraren herstellen, die für automatische Default-Instantiierungen registrierter Komponenten sorgen.

Das Problem von inkompatiblen Kompetenzen und Verhalten von versuchten Baustein-Verdrahtungen bzw. Konstruktionen lässt sich durch einen zwischengeschalteten `registrar_intermediate` lösen. Dieser kennt alle möglichen `adaptor_*` und sonstigen Konversions-Baustein-Typen samt ihren Eigenschaften und veranlasst bei Bedarf die automatische Zwischenschaltung der geeigneten Komponenten. Auf ähnliche Weise lässt sich Netzwerk-Transparenz durch automatisierte Zwischenschaltung von `remote` erzielen.

Bei Implementierung geeigneter Strategien (z.B. [LS94]) lässt sich Fehlertoleranz durch automatisches Re-Instantiieren von ausgefallenen Bausteinen erzielen, oder in Kombination mit der Netzwerk-Transparenz zur dynamischen Lastbalancierung nutzen. Es sind unzählige weitere

<sup>44</sup>Bei sicherheitskritischen Anwendungen kann die Ausführung von Instantiierungs- und Konstruktor-Routinen durch die `control`-Instanz eine Gefährdung darstellen, insbesondere wenn den in der nachgeschalteten `mmu_*`-Instanz ablaufenden Aktivitäten oder dem Inhalt von `code` nicht zu trauen ist. In diesem Fall sollte der Aufruf durch LRPC stattfinden und im Kontext der nachgeschalteten `mmu_*` ausgeführt werden.

Anwendungen für `strategy_*` und `registrar_*` denkbar.

Für Datenbank-Konstrukteure könnte eine reizvolle Aufgabe darin bestehen, einen Baustein `strategy_sql` zu entwickeln, der sich bei geeigneter Auslegung nicht nur für die klassischen Einsatzgebiete in Datenbanken eignet<sup>45</sup>, sondern das Prinzip der Trennung von logischen und physischen Zugriffspfaden auch auf solche Einsatzgebiete ausdehnt, die bisher als Domäne von Betriebssystemen<sup>46</sup> oder verteilten Systemen betrachtet wurden (insbesondere das Management virtueller Benutzer-Adressräume<sup>47</sup>).

<sup>45</sup> Durch zwischen geschaltete `remote`-Bausteine dürften sich auch bei den klassischen Einsatzgebieten von Datenbanken neue Konfigurationsmöglichkeiten ergeben, die bei bisherigen Datenbank-Architekturen nicht einfach zu realisieren waren. Zu nennen ist hier insbesondere die Einbeziehung des Verhaltens von Flaschenhälsern wie `remote`-Instanzen in rechnerübergreifende Query-Optimierungs-Strategien; es kann beispielsweise in manchen Szenarien durchaus vorteilhaft sein, Tabellen erst mittels `remote` auf Client-Rechner zu exportieren und dort mittels `buffer` zu cachen, bevor umfangreiche Joins oder andere Produkte daraus hergestellt werden.

<sup>46</sup> Durch derartige Verbindungen könnte man beispielsweise Pfadnamen durch SQL-Abfragen ersetzen.

<sup>47</sup> Dieser Ansatz geht über die Integration von Dateien in Datenbanken (vgl. [HR01, Abschnitt 3.5]) weit hinaus, da man sowohl Dateien als auch Datenbank-Datensätze jeweils sowohl über SQL als auch über Verzeichnis-Pfade zugreifbar machen kann; bei dieser Stufe der Integration verschwinden die konventionellen Unterschiede zwischen Datenbanken und Betriebssystemen völlig.

# Kapitel 5

## Callbacks durch **notify**-Operationen

Hierarchische Beziehungen drücken ein Abhängigkeitsverhältnis aus. Baustein-Instanzen, die „höher“ in einer Hierarchie stehen, haben „Macht“ oder „Kontrolle“ über solche, die weiter unten stehen. Die Leitungen des Baustein-Modells drücken derartige Hierarchie-Beziehungen aus.

Es gibt jedoch Fälle, in denen sich eine eindeutige Hierarchie nicht immer reinrassig ausführen lässt. Betriebssystem-Konstrukteure versuchen traditionell, solche Fälle möglichst zu vermeiden. Im Idealfall sollte ein Betriebssystem durch ein Schichtenmodell beschreibbar sein (vgl. [Dij71]). Die **notify**-Operationen sind dazu gedacht, um ein Schichtenmodell auch dann noch beizubehalten zu können, wenn es aus irgendwelchen Gründen klemmen würde (vgl. [Cla85]). In dieser Sichtweise haben **notify**-Operationen einen *Ausnahmecharakter*, der nicht missbraucht werden sollte. Ich werde versuchen, diejenigen Anwendungsfälle zu begründen, in denen ich Vorteile durch den Einsatz von **notify**-Operationen sehe.

### 5.1 Eigentums- und Besitzverhältnisse

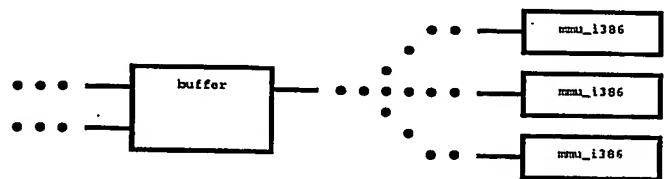
Die Begriffe *Eigentum* und *Besitz* stammen aus dem Rechtswesen. Dort kennzeichnen sie den Erstreckungsbereich eines Rechtes: wer *Eigentümer* einer Sache ist, der hat weitergehende Rechte als der *Besitzer*. Es ist insbesondere möglich, Sachen *auszuleihen* oder *zu vermieten*; dabei bleibt man weiterhin Eigentümer, obwohl der Besitzer (kurzzeitig) wechselt. Der neue Besitzer kann transitiven *Unterbesitz* an einer Sache oder Teilsache an ein oder mehrere *Unterbesitzer* weitergeben; diese *schulden* die Rückgabe an den *Hauptbesitzer*, das ist derjenige, von dem sie den Unterbesitz erhalten haben; dieser schuldet wiederum die Rückgabe des Besitzes an den Eigentümer. Der Eigentümer ist jemand, der niemandem die Rückgabe des Besitzes schuldet<sup>1</sup>. Im realen Leben kommen solche Rechtsverhältnisse z.B. bei der Vermietung und Untervermietung von Wohneigentum vor. Ein Mieter einer 5-Zimmerwohnung kann z.B. jedes Zimmer einzeln an Studenten untervermieten.

Diese Analogie möchte ich zur Darstellung der Verhältnisse in einem Betriebssystem einsetzen; dabei werde ich als Beispiel das Eigentum und den Besitz an Datenblöcken und an Locks nehmen; diese Methodik ist auch auf andere

Objekte übertragbar.

Eigentums- und Besitzverhältnisse sind von einer Hierarchie unabhängig<sup>2</sup> und können sich dynamisch zur Laufzeit ändern. Eine Baustein-Hierarchie teilt die Gesamtverantwortung eines Betriebssystems in statische Zuständigkeiten auf; bei der Übertragung eines Besitzes wird dagegen die temporäre Verantwortung für die zu verwaltenden Objekte innerhalb der Hierarchie weitergegeben.

Bei physischen Datenblöcken gibt es einen *natürlichen Eigentümer*: das ist diejenige Instanz, die die physische Adresse festlegt und verwaltet, z.B. indem sie den Datenblock „erzeugt“, d.h. in Existenz bringt<sup>3</sup>. In der Regel stellt dieser natürliche Eigentümer den Datenblock her, um ihn anschließend zu „verleihen“, d.h. das *Zugriffs- oder Besitzrecht* an eine andere Instanz für eine bestimmte Zeit abzutreten. Der neue Besitzer kann hierarchisch tiefer oder höher stehen. Unabhängig von der Hierarchie aber muß er den Besitz irgendwann wieder an den Eigentümer zurückgeben, ansonsten droht das bekannte Problem der Ressourcenverknappung durch nicht mehr recyclbare Ressourcen. Das folgende Bild zeigt einen Ausschnitt aus einem Szenario eines Rechners, der mit MMU-Hardware ausgestattet ist:



Prinzipiell kommt als Eigentümer oder Hauptbesitzer von Datenblöcken jede der beteiligten Baustein-Arten in Frage. Es bringt jedoch Vorteile, wenn die Adressvergabe der physischen Datenblöcke von der Adressvergabe im jeweiligen logischen Adressraum getrennt wird. Bei den logischen Adressen gibt es einen notwendigen Eigentümer bzw. Hauptbesitzer, der darüber zumindest in den Grundzügen bestimmen können muss: das ist der Benutzer, der ganz hinten als Endkonsument sitzt und bestimmen möchte, was

<sup>2</sup>Beispielsweise kann auch in einer absoluten Monarchie ein Untertan Eigentum erwerben, auf das der König keinen Zugriff hat. Wenn dies nicht möglich wäre, müssten die Untertanen verhungern, da sie ohne Zustimmung des Königs kein Eigentum und keinen Besitz an Lebensmitteln erwerben könnten, auch nicht an solchen Lebensmitteln, die sie selbst hergestellt haben.

<sup>3</sup>Beim Start des Betriebssystems gehört sämtlicher Speicher einem Urbesitzer, der ihn dann an andere Instanzen wie Caches oder `device_ramdisk` „ausleihen“.

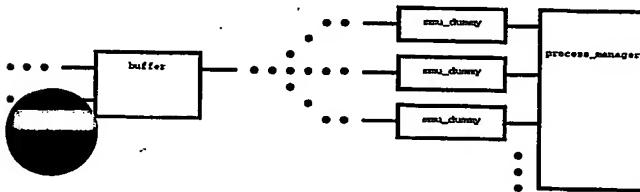
<sup>1</sup>Die einzige Ausnahme ist der Tod eines Eigentümers.

- der Rechner und das Betriebssystem für ihn tun soll; er kann zwar einige dieser Aufgaben an den Compiler oder das Laufzeitsystem (Benutzer-Heap-Verwaltung) delegieren, aber letztlich ist er der Herr über seinen virtuellen Adressraum.

Physische Adressen lassen sich prinzipiell sowohl ganz oben bei den `mmu_*`-Instanzen als auch ganz unten bei der `buffer`-Instanz bzw. der evtl. vorgeschalteten `device_mem`-Instanz verwalten. Bei Vorhandensein von MMU-Hardware ist es jedoch aus mehreren Gründen vorteilhaft, die Verwaltung physischer Adressen und damit das Eigentumsrecht der `buffer`- bzw. `device_mem`-Instanz und nicht den vielen `mmu`-Instanzen zu übertragen:

- Verschiedene `mmu`-Instanzen greifen häufig auf dieselben Datenblöcke zu; beispielsweise gemeinsamer Maschinencode. Wenn das Eigentumsrecht bei der `buffer`-Instanz liegt, ist die gemeinsame Nutzung von Datenblöcken relativ einfach zu realisieren. Billigt man das Eigentum dagegen den `mmu`-Instanzen zu, dann findet entweder keine gemeinsame Nutzung statt, oder diese muss wesentlich aufwendiger z.B. über Cache-Kohärenz-Protokoll hergestellt werden, das sich über die gesamte Hierarchie hinweg erstrecken kann.
- Die Gefahr von Deadlocks ist leichter vermeidbar, da eine zentrale Instanz besser die Übersicht über die vergebenen Ressourcen wie Locks behalten kann.

Im folgenden Beispiel ist keine MMU-Hardware vorhanden. Die Verwaltung der physischen Datenblock-Adressen muss(!) wegen der notwendigen Übereinstimmung von logischen und physischen Adressen bei den `mmu_dummy`-Instanzen erfolgen, die als Ersatz für die fehlende MMU-Hardware stehen:



Wenn keine MMU vorhanden ist, dann ist eine unabhängige Übersetzung von logischen Adressen nach physischen Adressen nicht möglich; es bleibt als einzige Lösung, die physischen Adressen mit den logischen Adressen gleichzusetzen. Daher ist es günstiger, das Eigentumsrecht zumindest an denjenigen Datenblöcken, die in Prozessabbildern verwendet werden, ganz oben in der Hierarchie bei `mmu_dummy` oder `process_manager` zu verwalten (andernfalls könnten keine Transformationen durch dazwischenliegende `dir_*`-Instanzen durchgeführt werden, oder es müssten zusätzliche Performanz fressende Kopien erstellt werden<sup>4</sup>).

<sup>4</sup>Wenn man Kopien herstellt, dann betreibt man letztlich eine Mischform der Zuordnung von Eigentumsrechten an verschiedene Instanzen. Die „Originale“ werden unten in der Hierarchie verwaltet (siehe MMU-freies Ur-Unix mit Buffer-Cache in [Lio96]), die Kopien entstehen dann implizit z.B. bei read- oder write-Operationen.

Diese Erörterung sollte bewusst gemacht haben, dass Eigentums- und Besitzverhältnisse prinzipiell unabhängig von Hierarchie-Beziehungen sind, und dass eine gut durchdachte Zuordnung Implementierungs-Aufwand und Performance sparen kann.

Die in Abschnitt 3.1 eingeführte Unterscheidung zwischen logischem und physischem IO auf Nestern korrespondiert mit dem Konzept der Eigentums- und Besitzverhältnisse in folgender Weise:

- Ein Eigentümer bzw. Hauptbesitzer bietet stets logischen IO an seinem Ausgang an. Die höheren Hierarchie-Instanzen sind gezwungen, logischen IO zu verwenden, wenn sie in den (temporären) Besitz eines Datenblocks gelangen wollen; der in Abschnitt 3.3.3 eingeführte Referenzzähler dient zur Verwaltung der „Ausleih-Rechtsverhältnisse“; im juristischen Kontext würde man dies als „Schuldverhältnis“ charakterisieren. Ein „Schuldner“ muss irgendwann seine „Schulden“ wieder „zurückzahlen“; dies geschieht in der hier vorgestellten Architektur durch korrekte Paarung von `put`.
- Ein Eigentümer betreibt an seinem Eingang ausschließlich physischen IO; die unterliegenden Instanzen brauchen nur diese Betriebsart zu unterstützen.

Es ist jedoch sinnvoll, dass zwischenliegende Instanzen wie `dir_*` sowohl die Kompetenzen zu logischem als auch zu physischem IO haben sollten. Dies ermöglicht nicht nur den Einsatz der neuen Architektur in MMU-freier Hardware wie *embedded systems*, sondern rein theoretisch sogar den gemischten Einsatz von solchen Prozessen, die in virtuellen Adressräumen laufen, neben MMU-übersetzungsfreien Prozessen<sup>5</sup>.

## 5.2 Rückforderung von Eigentum

Wie im realen Leben kann es auch in einem Betriebssystem vorkommen, dass ein „Schuldner“ seine „Schulden“ nicht bezahlt, oder nicht „rechtzeitig“ bezahlt. Die Einführung eines „Mahnwesens“ ist prinzipiell geeignet, den Systemdurchsatz zu verbessern oder in hoffnungslosen Mangelsituationen überhaupt noch ein Weiterarbeiten zu ermöglichen.

```

notify_data(nest, log_addr, len,
            phys_addr, urgency) → success
notify_lock(nest, log_addr, len,
            try_address, try_len,
            kind, urgency) → success
  
```

Diese Operationen durchlaufen eine Baustein-Hierarchie in umgekehrter Richtung, d.h. vom Eigentümer hin zu den Konsumenten. Die Konsumenten werden gebeten oder gezwungen, die jeweilige Ressource (ein Datenblock mit der angegebenen logischen und physischen Adresse, oder ein

<sup>5</sup>Wieviel Sinn dies macht, ist eine andere Frage. Theoretisch kann ein MMU-freier Prozess schneller laufen, da der Hardware-Aufwand zur Adressübersetzung wegfällt, und garantiert keine Verzögerungen durch Seitenfehler-Unterbrechungen auftreten. Letzterer Effekt wird in echtzeitfähigen Betriebssystemen jedoch auch durch vorgeladene und nicht auslagerbare Speicherseiten erzielt.

Lock der angegebenen Art) zurückzugeben. Der Parameter `urgency` kann einen der folgenden Werte annehmen:

- `ask` Es wird lediglich angefragt, ob eine Rückgabe der Ressource ohne größere Mühen und Störung des Betriebes möglich wäre. Der Empfänger braucht außer der boolschen `success`-Rückmeldung nichts zu unternehmen.
- `try` Der Empfänger wird gebeten, die Ressource zurückzugeben, wenn es den Betrieb nicht unverhältnismäßig stört. Ein genauer Zeitpunkt für die Rückgabe wird nicht vorgeschrieben.
- `command` Der Empfänger wird aufgefordert, die Rückgabe unverzüglich (d.h. noch vor der `success`-Rückmeldung) vorzunehmen. Eine Verweigerung der Rückgabe durch Rücklieferung von `success=false` ist nur dann statthaft, wenn der Empfänger wegen der fehlenden Ressource seine Tätigkeit ganz einstellen müsste.
- `force` Der Empfänger muss die Ressource unverzüglich zurückgeben, auch wenn er dadurch zum Abbruch seiner Tätigkeit gezwungen wird. Falls er sich dennoch weigern sollte<sup>6</sup>, werden alle seine Ressourcen „konfisziert“, d.h. vom Eigentümer so behandelt, als wären sie zurückgegeben worden, und es finden im Zukunft keine Operations-Ausführungen mehr statt, die durch die Empfänger-Instanz in Auftrag gegeben worden sind oder noch gegeben werden.

Die `notify_*`-Operationen haben den Charakter einer *Ausnahme* und sind konzeptuell mit den Signalen von Unix und anderen Betriebssystemen vergleichbar. Prinzipiell sind daher *Wettrennen* zwischen `notify_*`-Operationen und Anforderungs- bzw. Rückgabe-Operationen des Besitzers möglich. Diese Wettrennen treten auch bei vergleichbaren Situationen in konventionellen Betriebssystemen auf und sind meiner Ansicht nach prinzipiell (siehe Diskussion in Anhang C). Wettrennen lassen sich zwar durch architekturelle Maßnahmen prinzipiell vermeiden (vgl. Anhang C), jedoch geht dies i.a. zumindest bei verteilten Systemen zu Lasten der erzielbaren Parallelität und Performance, da zumindest dort das Wettrennen-Problem inhärent ist und ohnehin gelöst werden muss.

Eine vom Wettrennen-Problem unabhängige Frage betrifft die „rechtmäßige“ Ausübung „brutaler Gewalt“, insbesondere der `force`-Modus. Es versteht sich von selbst, dass ein Eigentümer oder Hauptbesitzer große Macht über jeden ausüben kann, der etwas von ihm als Unterbesitz erhalten hat, da er auch dann auf Rückforderung bestehen kann, wenn dadurch die Existenz des Unterbesitzers gefährdet wird. Da die Besitzschuldverhältnisse prinzipiell unabhängig von den in Abschnitt 2.7 besprochenen Zugriffsrechten

<sup>6</sup>Wenn der Empfänger z.B. in eine Endlosschleife geht oder z.B. bei der Ausführung der `notify_*`-Operation in einen Deadlock gerät oder wenn z.B. im RPC-Modell der Bearbeiter-Kontrollfluss verstirbt, kann eine „Weigerung“ auch schlicht darin bestehen, dass nichts geschieht. Da interne Angelegenheiten eines Bausteins nicht von außen „aufgemischt“ werden sollten, bleibt zur Lösung dieses Problems letztlich nur das Setzen einer Zeitschranke, deren Ablauf als „Weigerung“ interpretiert wird.

sind, könnte man die Frage stellen, ob diese Machtstellung insbesondere von hierarchisch untergeordneten Instanzen nicht unangemessen sein könnte. Hierauf gibt es eine prinzipielle Antwort: Nein. Wer (auf welche Weise auch immer) einmal die Macht erhalten hat, bestimmte Ressourcen verwalteten zu dürfen, der kann von dieser Macht bereits bei der Ressourcenvergabe unangemessenen Gebrauch machen. Er kann rein theoretisch bereits zu diesem Zeitpunkt „bösartig“ handeln und eventuelle Zugriffsbeschränkungen umgehen; ein typisches Beispiel ist die Hauptspeicherverwaltung, die bei Kompromittierung sämtliche Schutzmechanismen des gesamten Systems aushebeln kann, die auf ihr basieren<sup>7</sup>. Daher muss man einem Ressourcenbesitzer im Hinblick auf die Einhaltung geforderter Zugriffsrechts-Beschränkungen ohnehin vertrauen. Das Recht zur Rückforderung benötigt in dieser Sichtweise kein zusätzliches Vertrauen. Wenn man einer Instanz nicht vertrauen kann, dann darf man sie von vornherein nicht mit der Verwaltung von Eigentum oder Unterbesitz beauftragen (z.B. indem man für eine geeignete Aufteilung der Macht sorgt).

Es gibt weitere Gründe, die den Entzug beliebiger Ressourcen zu einem Pflicht-Mechanismus in zukunftssicheren Betriebssystemen machen. Großrechner unterstützen seit langem das sogenannte „Hot-Plugging“ von Hardware-Komponenten, also den Austausch mitten im laufenden Betrieb. Dies betrifft nicht nur CPUs, die in fast allen Betriebssystemen entziehbare Ressourcen darstellen, sondern auch Speicher-Module. Die Problematik des Laufzeit-Austausches taucht fernerhin im IO-Bereich wie z.B. bei RAID auf; bei sicherheitskritischen Anwendungen wird ein prophylaktischer Austausch von Hauptspeicher-Modulen manchmal sogar turnusmäßig durchgeführt. Die Fähigkeit zum Austausch im laufenden Betrieb erfordert einen Mechanismus, mit dem der Entzug von Ressourcen in geordneten Bahnen durchgeführt werden kann.

Microsoft hat das Konzept der *opportunistischen Locks* [OPL] in die Betriebssystem-Schnittstelle der neueren Windows-Versionen integriert und damit zumindest für einige Anwendungen auch Anwendungsprogrammen verfügbar gemacht. Opportunistische Locks<sup>8</sup> sind dazu gedacht, um das Cache-Kohärenz-Problem in verteilten Systemen durch den *gezielten* Entzug von Ressourcen zu beschleunigen; wenn kein aktueller Bedarf für einen Entzug vorhanden ist, dann wird auf das zeitaufwendige Durchreichen der Ressource über das Netzwerk verzichtet („aggressives Caching“). Im hier vorgestellten Modell wird der Entzug mittels brutaler Gewalt nur als *letztes Mittel* betrachtet; für das im nächsten Abschnitt näher behandelte Cache-Kohärenz-Problem wird der `try`-Modus favorisiert. Ein weiterer grundlegender Unterschied der hier favorisierten spekulativen Locks zu opportunistischen Locks

<sup>7</sup>Sofern sie nicht durch andere Instanzen wie z.B. Wächter daran gehindert wird. Dies ist jedoch kein Gegenargument, da eine Wächter-Instanz konsistenterweise auch die Rückforderungen überwachen sollte, so daß unter dem Strich eine Aufteilung der Macht auf mehrere Instanzen erfolgt (gegenseitige Überwachung).

<sup>8</sup>Microsofts opportunistische Locks stellen eine eigene Locking-Art dar, die von anderen Locking-Arten unabhängig sind. Dies erhöht nicht nur die Komplexität der Schnittstellen, sondern kann z.B. zu der Situation führen, dass eine maliziöse Anwendung, die einen opportunistischen Lock entzogen bekommen hat, mittels regulärer Locks das System dennoch lahm legen kann.

- besteht darin, dass auf Rückforderungen im `try`-Modus nicht mit der Rückgabe des *gesamten* gesperrten Bereichs, sondern auch nur eines Teiles reagiert werden kann; dies ist insbesondere in Kombination mit der spekulativen Anforderung von vergrößerten Adress-Bereichen durch die `try_address`- und `try_len`-Parameter vorteilhaft (vgl. Abschnitt 3.3.5). Die Unterscheidung zwischen einem Kernbereich und einem spekulativ erweiterten Bereich wird auch in `notify_lock` getroffen; damit kann der Empfänger einer Rückforderung selbst entscheiden, wieviel er ohne größere Behinderung seiner Arbeit zurückgeben kann.

### 5.3 Synchronisation zwischen Kopien: spekulative Locks

Verteilte Systeme unterscheiden sich aus Sicht der hier vorgestellten Architektur hauptsächlich<sup>9</sup> durch eine Eigenschaft von einem alleinstehenden Rechner: das ist die Zugriffs- bzw. Latenzzeit der `remote`- und `mirror`-Befehle, sowie die maximale Datenrate, die ihren Durchsatz begrenzt. Dieses Problem tritt in Gestalt der bekannten „Flaschenhälse“ auch in alleinstehenden Systemen auf und ist als so genanntes *Flaschenhals-Problem* lange bekannt, das sich z.B. bei virtuellem Speicher als *Thrashing* (vgl. [DS72]) äußern kann. Die spezifische Problematik steckt bei verteilten Systemen lediglich im *Ort* (Hierarchie-Ebene), an dem sich der Flaschenhals befindet.

Zur Lösung des Flaschenhals-Problems gibt es zwei erprobte grundlegende Strategien: man versucht den Flaschenhals durch Hardware-Aufrüstung zu erweitern, und/oder die zur Lösung einer Aufgabe notwendige Belastung zu senken. Während sich bei der Hardware klassischer Peripheriegeräten in der Vergangenheit ständige Fortschritte sowohl bei den Latenzzeiten als auch beim Durchsatz ereignet haben und voraussichtlich weiter ereignen werden, sind derartige Fortschritte bei räumlich weit verteilten Systemen nur beim Durchsatz, nicht hingegen bei der Latenzzeit möglich (vgl. [TL93]), da die Kommunikation der Lichtgeschwindigkeit begrenzt wird. Daher ist unvermeidbar Datenverkehr bzw. unnötiges Warten auf die Ausführung von Operationen bei verteilten Systemen noch dringender zu vermeiden als in alleinstehenden Systemen. Die hier vorgestellte Architektur legt auch aus diesem Grund besonderen Wert auf asynchrone Kommunikation.

Wie bereits in Abschnitt 4.1.2 (Baustein `buffer`) dargestellt, besteht der Zweck eines Caches in der Entkopplung von Flaschenhälsen und der Reduktion der darüber laufenden transfer-Operationen. Während bei alleinstehenden Rechnern Konfigurationen mit nur einer zentralen `buffer`-Instanz möglich sind, ist der Einsatz kaskadierter verteilter Caches (`buffer`-Instanzen) bei verteilten Systemen aus Performanz-Gründen de facto unumgänglich. Kaskadierte Caches führen zwangsläufig zum bekannten Cache-Kohärenz-Problem.

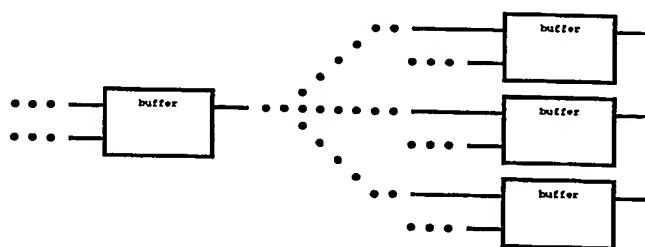
<sup>9</sup>Das häufig angeführte Problem der *Ausfallsicherheit* sehe ich nicht als spezifisch für verteilte Systeme an, obwohl eine brauchbare Lösung dort von besonderer Wichtigkeit ist. Konzepte zur Erzielung von Ausfallsicherheit sollten für alle Bausteine uniform sein, unabhängig von der konkreten physischen Verteilung.

Das Cache-Kohärenz-Problem lässt sich in der Systematik der hier verwendeten Begriffe auf folgende Weise formulieren: es sind mehrere Besitzer derselben logischen Ressource vorhanden.

Im Rechtswesen taucht dieses Phänomen in den Begriffen *Gemeinschaftseigentum* (z.B. bei Wohnungseigentümer-Gesellschaften) und *gemeinschaftlicher Besitz* (z.B. Anmietung durch einen Verein) auf. Grundsätzlich sind mehrere Besitzer gleichberechtigt, und sie müssen sich bei allen Fragen betreffs des Besitzes gegenseitig *auseinandersetzen*; dies geschieht im sogenannten *Innenverhältnis*. Der Grund hierfür ist, dass ein Besitz exklusiv nutzbare Eigenschaften haben kann, so dass eine gleichmäßige und/oder gleichzeitige Nutzung durch alle Besitzer nicht immer möglich ist.

Die Auseinandersetzung über exklusiv nutzbare Ressourcen findet in traditionellen alleinstehenden Betriebssystemen bevorzugt durch Delegation der Verwaltungsaufgaben an eine zentrale Instanz (z.B. ein Lock-Manager) statt. Diese Lösung hat den Vorteil hoher Einfachheit bei geringen Kosten, da durch die Delegation praktisch vernachlässigbare Latzenzeiten entstehen. In einem verteilten System ist dagegen eine Delegation an eine einzige Instanz i.a. wegen der Kommunikations-Latzenzen sehr kostenintensiv.

Man kann sich Worst-Case-Zugriffsszenarien ausdenken, bei denen die Latenz-Kosten eines verteilten Systems zwangsläufig bei jedem einzelnen Zugriff entstehen müssen, weil sie wegen einer „genügend bösartigen“ Aufgabenstellung nicht zu umgehen sind (analog zum Working-Set-Modell). Dies trifft übrigens auch für Caches alleinstehender Systeme zu, die man ebenfalls mit solchen Zugriffsmustern belasten kann, dass sie den bekannten *Thrashing-Effekt* zeigen, bei dem sie ihre Wirksamkeit fast vollständig einbüßen können. Bei alleinstehenden Systemen gilt die lange bekannte Grundregel, dass Thrashing nur dann vermeidbar ist, wenn der *Working Set* [Den68] der Last in den Cache passt. Diese lange bekannte Grundregel gilt aber auch in einem verteilten System mit mehreren Caches, allerdings in nochmals verschärfter Form. Schauen wir uns dazu das folgende Szenario aus mehreren örtlich weit verteilten Caches an, die ich „hierarchisch hochstehende Caches“ nenne, die an einen gemeinsamen Basis-Cache angeschlossen sind:



Damit überhaupt ein Entkopplungs-Effekt durch die hochstehenden `buffer`-Instanzen entstehen kann, müssen ihre Working-Sets, bezogen auf die zeitliche Größenordnung der Kommunikations-Latzenzen, „einigermassen disjunkt“ sein. Andernfalls tritt analog zu [DS72] zwangsläufig ein Thrashing-Effekt zwischen den hochstehenden `buffer`-Instanzen ein, der in der hier gewählten Hierarchie-Struktur

über die untenstehende Basis-buffer-Instanz abgewickelt wird und auch durch Vergrößerung der Caches nicht eliminierbar ist, wie Denning sehr einleuchtend bereits vor langer Zeit gezeigt hat. Gegen diese Art von Thrashing hilft meiner Ansicht nach überhaupt kein Kraut: es ist verteilten Systemen inhärent, dass modifizierte Daten irgendwie an den Ort der anderen buffer-Instanz gebracht werden müssen, wenn sie dort von der Aufgabenstellung her unbedingt benötigt werden, und das kostet zwangsläufig Latenzzeit (weil die Lichtgeschwindigkeit als Naturkonstante nicht umgehbar ist). Ich sehe daher keine andere Möglichkeit, als die Aufgabenstellung so zu gestalten, dass ein derartiges Thrashing nicht stattfindet. Dazu ist eine notwendige Bedingung, dass die Working-Sets an verschiedenen Orten einigermaßen disjunkt voneinander sein müssen.

Unter der Annahme, dass die Working-Sets der hochstehenden buffer-Instanzen disjunkt sind, sieht die Lösung des Problems folgendermassen aus: es ist nichts anderes zu tun, als die vorkommenden Working-Sets einigermaßen genau zu bestimmen. Dies macht ein ausschließlich von Anforderungen getriebener Cache von Natur aus ganz von selbst. Anders sieht die Situation höchstens dann aus, wenn das Laden der oberen Cache-Instanzen mit Daten zusätzlich auf *spekulative* Weise erfolgt, z.B. beim sogenannten *Preloading*.

Zur Analyse des Preloadings sehen wir uns den Best Case an, bei dem wir die folgenden idealisierten Annahmen treffen:

1. Die Caches seien jeweils genügend groß, so dass alle insgesamt benötigten Datenmengen in jedem einzelnen Cache Platz haben.
2. Die Datenübertragungs-Rate sei näherungsweise unendlich groß, die Latenzzeit dagegen ein fester Wert echt größer Null.
3. Jegliche Änderung eines Datums werde *sofort* an alle anderen Cache-Instanzen propagiert (was wegen der unendlichen Datenübertragungs-Raten auch kein Problem darstellt); diese Propagation kostet näherungsweise nichts.
4. Welche Version eines Datums während der Kommunikations-Latenzzeiten an welcher Stelle zur Verfügung steht, wird idealisierend als ständig überall bekannt vorausgesetzt.

Trotz dieser bestmöglichen Annahmen, die unter der vorgegebenen räumlichen Verteilung nicht mehr zu verbessern sind, kann hier immer noch ein Thrashing zwischen den hochstehenden Caches stattfinden, sofern man nur die Anforderungen durch die Aufgabenstellung genügend „bössartig“ gestaltet. Ist diese Bösartigkeit jedoch nicht gegeben (weil die Working Sets aller Teilnehmer-Instanzen disjunkt sind), dann braucht *niemals* irgend eine buffer-Instanz auf eine andere zu warten. Damit haben wir den Best Case dessen, was mit Hilfe von Caching-Strategien überhaupt erreichbar ist, nämlich die totale Entkopplung aller Aktivitäten und damit die Illusion, dass Latenzzeiten nicht vorhanden seien.

Da die getroffenen Annahmen in der Praxis natürlich nicht zutreffen<sup>10</sup>, lautet die Frage, wie man den Best Case approximieren kann. Für die Annahme 1 gibt es lange bekannte und erprobte Strategien der Adaption von Caches an die Working-Sets der Anforderungen, insbesondere LRU und LFU. Zur Lösung des Problems 2 wird insbesondere in Hardware-Architekturen (beispielsweise in Prozessor-Cache-Hierarchien) die *explizite Priorisierung* der Datentransfers eingesetzt: beim Transport durch einen Flaschenhals müssen solche Transport-Aufträge dringender behandelt werden, von deren Ausführung andere Aktivitäten abhängen; dagegen sollten spekulativen Transport-Aufträge unbedingt geringer priorisiert werden, um dringendere Aktivitäten nicht durch Verstopfung des Flaschenhalses zu behindern. Dies wird bei der hier vorgestellten Architektur durch die IO-Prioritäten ermöglicht (siehe Abschnitt 2.8.3). Das Problem 3 lässt sich dadurch lösen, dass jegliche Änderungen grundsätzlich *sofort*<sup>11</sup> durch Erzeugen eines IO-Auftrages (*transfer*-Operation) bekannt gemacht werden, allerdings standardmäßig nur mit Background-IO-Priorität, die nur bei Vorliegen höherer Dringlichkeit erhöht wird. Das Problem 4 erfordert eine genauere Analyse:

Wie bereits in Abschnitt 3.3.3 dargestellt und in Abschnitt 5.1 begründet, erfordert die Verwaltung von Besitz eine transiente Zuordnung des Besitz-Status (z.B. als Referenzzähler, ggf. auch der Versions-Zuordnung). Dieser Status ist an den Ort gekoppelt, für den der jeweilige (Unter-)Besitz-Status verwaltet wird. Da alle hochstehenden Caches aus Sicht des Basis-Cache Konsumenten sind, müssen sich diese wie alle anderen parallel arbeitenden Konsumenten an ein *multi*\*-Verhalten halten, um die Determinanz für ihre jeweiligen Konsumenten sicherstellen zu können. Daraus folgt insbesondere, dass Locks prinzipiell bis auf den Basis-Cache durchgereicht werden müssen, ansonsten könnten die Verhaltens-Modelle verletzt werden. Wenn man jeden Lock einzeln durchreichen würde, dann entstünde jedesmal eine Wartezeit durch die Kommunikations-Latenzen, was sich katastrophal auf den Durchsatz auswirken würde. Diese Pflicht zum Durchreichen ist jedoch eine *Mindest-Anforderung*, die auch *übererfüllt* werden darf. Die Idee besteht nun darin, dass bei der erstmaligen Anforderung eines Locks nicht die vom Endkonsumenten angeforderte Granularität durchgereicht wird, sondern möglichst ein *spekulativ vergrößerter* Adressbereich (z.B. ein gesamtes Unter-Nest oder ein gesamtes Verzeichnis-Nest). Ein dazu geeigneter Mechanismus ist bereits in Abschnitt 3.3.5 vorgestellt worden (Parameter *try\_address* und *try\_len*). Wenn es einem der hochstehenden Caches gelungen ist, den Working-Set seiner Konsumenten einigermaßen realistisch spekulativ einzuschätzen, dann braucht er spätere Lock-Anforderungen seiner Konsumenten nicht jedesmal Zeit raubend zum Basis-Cache durch zu schalten, sondern kann den bereits erhaltenen spekulativen Besitz ohne Wartezeit an seine Konsumenten als Unterbesitz weiter geben, sobald diese eine entsprechende Anforderung stel-

<sup>10</sup>Die Fortschritte bei den Kommunikationstechniken gehen jedoch in die Richtung, dass die Best-Case-Annahmen immer besser durch verbesserte Technik angenähert werden.

<sup>11</sup>Damit ist nicht die Granularität einzelner Schreib-Maschinenoperationen gemeint, sondern eine „zeitnahe“ Meldung in der Größenordnung der Kommunikations-Latenzen.

len. Zu beachten ist dabei lediglich, dass spekulativ vorab angeforderte Lock-Arten die korrekte Semantik bei späterer Nachforderung durch die Endkonsumenten ergeben müssen; i.a. wird man dabei statt eines Read-Locks einen Write-Lock spekulativ anfordern müssen, um die Aktualität der neuesten Version garantieren zu können (ansonsten könnten andere Instanzen Veränderungen vornehmen, die zu einem unbemerkt Veralten der spekulativ angeforderten Version führen).

Spekulationen über den zukünftigen Working-Set der Endkonsumenten können insofern fehl gehen, als dass zu grossen Bereiche vom Basis-Cache angefordert wurden, die später doch noch von anderen hochstehenden Caches benötigt werden<sup>12</sup>. Fehlspirationen lassen sich durch Rückforderungen mittels `notify_*`-Operationen korrigieren. Der Basis-Cache kann grundsätzlich bei Anforderung bereits vergebener Locks entsprechende `notify_*`-Operationen an den/die momentane(n) Besitzer-Instanz(en) generieren. Falls er sich die Unterschiede zwischen vergebenen Kern- und Erweiterungsbereichen merkt (sowie ggf. von später hinzukommenden Kernbereichen durch den neuen Besitzer wah, aber asynchron informiert wird), kann er auf diese „aussichtslosen“ Rückforderungen, die vergebene Kernbereiche betreffen, auch verzichten. Rückforderungen von Kernbereichen sollten möglichst nur für solche Adressbereiche gestellt werden, die ein anderer Cache auch wirklich aktuell benötigt. Es ist Sache des Fehlspkulanten, ggf. auch gröbere Bereich als den aktuell zurück geforderten oder sogar gröbere als den vorgeschlagenen Erweiterungsbereich an den Basis-Cache zurück zu geben, um damit weiteren zukünftigen Rückforderungen spekulativ vorzubeugen. Eine Rückgabe ist darüber hinaus auch jederzeit spontan möglich, beispielsweise wenn erkannt wurde, dass ein bestimmter Bereich auf jeden Fall nicht mehr benötigt wird.

Insgesamt bewirkt diese Vorgehensweise eine spekulative Vorab-Verteilung von Ressourcen, die durch die aktuellen Anforderungen nachträglich korrigiert wird. Die Rückgabe spekulativ angefordelter Ressourcen kann durch verschiedene Strategien erfolgen. Eine mögliche Rückgabestrategie ist, den zwischen dem nächstliegenden tatsächlich genutzten Bereich und dem rückgeforderten Bereich liegenden spekulativ angeforderten Bereich zu halbieren, also die Hälfte weiter zu behalten, die andere Hälfte dem Rückforderer zu überlassen. Falls die Working Sets der Endkonsumenten disjunkt und näherungsweise zusammenhängend sind, dann ergibt sich dadurch im Worst Case eine logarithmische Anzahl von Rückforderungen (in Abhängigkeit von der Größe des aufzuteilenden Adressbereiches). Nach dieser höchstens logarithmischen Anzahl an Rückforderungen ist der vorab unbekannte genaue Verlauf der disjunktiven Working Sets empirisch festgestellt worden, und es finden keine weiteren Rückforderungen mehr statt (eingeschwungener Zustand des Systems). Nach dem Einschwingen sind keine weiteren Synchronisationen mehr erforderlich, sofern sich die Working Sets nicht ändern; damit ist der obige Best Case relativ gut approximiert. Bei „langsamen“

Änderungen der Working Sets finden zwar ab und zu noch Rückforderungen statt, diese haben jedoch ein relativ geringes Gewicht in den Gesamtzahl aller Operationen; damit ist hier ebenfalls der Best Case relativ gut approximiert. Bei zu starker Überlappung der Working-Sets bzw zu großer Änderungsrate kann es zu einem Thrashing kommen, das jedoch laut obiger Argumentation prinzipiell nicht vermeidbar ist.

<sup>12</sup>Falls die Annahme disjunkter Working Sets doch nicht zutreffen sollte, liegt bei der hier vorgestellten Betrachtungsweise die „Schuld“ nicht bei den Caches, sondern beim Verhalten der Endkonsumenten.

# Kapitel 6

## Generische Operationen

*Generische Operationen* sind Operationen, die über eine *polymorphe Schnittstelle* aufrufbar sind.

In Betriebssystemen wird das Paradigma der Generizität bzw. der Polymorphie bereits sehr lange eingesetzt; bekannte Beispiele finden sich u.a. bei polymorphen Kern-Schnittstellen wie `ioctl` in Unix, oder bei den Operations-Sprungtabellen von IBM-Betriebssystemen, die verschiedene Arten von Polymorphie bereits in den 1960er Jahren implementierten.

Bevor wir zur Untersuchung der Arten von Polymorphie kommen, möchte ich klären, was ein Operations-Aufruf überhaupt ist:

Ein *Operations-Aufruf* ist die Übergabe von *Information* von einer *Aufrufer-Instanz* an eine *Bearbeiter-Instanz*.

Nach dieser Definition ist die Rückgabe von Ergebnissen, z.B. in Ergebnis-Parametern, ebenfalls ein Operations-Aufruf, nur in umgekehrter Richtung, d.h. vom ursprünglichen Bearbeiter zurück zum ursprünglichen Aufrufer. In diesem Fall spricht man von einem *Operations-Aufrufs-Paar*.

Operations-Aufrufe bzw. -Paare lassen sich auf verschiedene Weisen realisieren. Bekannte Realisierungs-Arten sind z.B. Prozeduraufrufe mittels Übergabe von Parametern auf dem Stack eines Programmiersprachen-Laufzeitsystems, oder der RPC (Remote Procedure Call). Ersterer Fall lässt nur den *synchrone Aufruf*, letzterer Fall auch den *asynchrone Aufruf* zu. Weitere bekannte Realisierungs-Arten sind das Starten von Kontrollflüssen (asynchrones Verhalten), oder die Weitergabe der Flusskontrolle in einem Coroutinen-Modell durch Aufruf einer Transfer-Operation (dies bewirkt *synchrone* Verhalten aus Sicht des Gesamtsystems).

Ich möchte auf eine weitere bekannte Art der Realisierung von Operations-Aufrufen fokussieren: das so genannte *Nachrichten-Paradigma*<sup>1</sup>, das zwar oft im Kontext von CSP (siehe [Hoa78]) in Erscheinung tritt, prinzipiell jedoch unabhängig von sequentiellen Kontrollflüssen ist. Das Nachrichten-Paradigma wird grundlegend in der Hardware verwendet und steckt z.B. hinter den weit verbreiteten read-write-Schnittstellen, bei denen ja auch Informationen von

<sup>1</sup>Nachrichten dienen zur Übertragung von Information (umgekehrt kann jedoch auch das Nicht-Senden oder das Nicht-Ankommen einer Nachricht eine Information darstellen, z.B. beim Ablauen eines Timeouts; vgl. [Lam84]). Eine Übertragung von Information muss nicht unbedingt durch Herstellen einer *Kopie* erfolgen; beispielsweise stellt auch die Nutzung eines gemeinsamen Puffers durch mehrere Instanzen eine Realisierung einer Nachrichten-Übertragung dar.

einer Instanz an eine andere weitergegeben werden.

Man sollte sich unbedingt klar machen, dass zwischen einer reinen Nachrichten-Übertragung und einem Operations-Aufruf *überhaupt kein prinzipieller Unterschied* besteht. Ein solcher Unterschied entsteht erst durch die *Interpretation* der Nachricht durch den Empfänger, sowie ggf. durch seine *Reaktion* auf die interpretierte Nachricht.

Dies bedeutet für unsere Betriebssystem-Architektur, dass wir zur Realisierung von generischen Operationen keine neuen Konzepte einführen müssen, sondern die bereits ausführlich vorgestellten Abstraktionen Nest und Baustein verwenden können.

Konkret wird dies z.B. auf folgende Weise realisiert: Einer Nest-Instanz  $s$  wird ein so genanntes *Operations-Nest* als dynamisches Attribut zugeordnet, oder mit Hilfe einer eigenen Elementarfunktion `get_ops(s)`. Das Operations-Nest dient zum Austausch der Nachrichten für die generischen Operationen, die auf dem *zugeordneten Nest*  $s$  ausgeführt werden sollen. Zu einem Operations-Nest  $op$  kann man umgekehrt das zugeordnete Nest durch eine Elementarfunktion `get_nest(op)` zugänglich machen. Es gilt dann `get_nest(get_ops(s)) = s`.

Zum Austausch der Nachrichten werden die in Abschnitt 3.3.8 behandelten Elementaroperationen `get_address`, `put_address`, `get`, `put`, `transfer` und `wait` verwendet. Das Operations-Nest darf zu einem beliebigen Zeitpunkt mehrere Operations-Aufrufe enthalten und ist daher prinzipiell geeignet, um die Funktionalität von Auftrags-Warteschlangen, Gerätetreiber-Warteschlangen u.ä. auszuführen.

Zum Übertragen der Nachrichten in einem Operations-Nest müssen auf jeden Fall Elementaroperationen verwendet werden<sup>2</sup>, die an einen Schnittstellen-Mechanismus aus Abschnitt 2.3.2 gebunden sind, und mit deren Hilfe erst das Konzept der generischen Operationen auf einer höheren Abstraktionsstufe implementiert wird. Rein theoretisch wäre es möglich, auf die Implementierung der Elementaroperationen im zugeordneten Nest zu verzichten und diese ausschließlich durch generische Operationen im zugehörigen Operations-Nest auszuführen. Nicht nur aus Performanz-Gründen, sondern auch aus Symmetriegründen propagiere ich die umgekehrte Methodik, dass Elementaroperationen auf *beide* Arten aufrufbar sein sollen (und dabei dasselbe tun sollen). Es spielt dann keine Rolle, ob Elementaropera-

<sup>2</sup>Dies ließe sich durch Benutzung von `get_ops(get_ops(s))`, also Einführen eines Operations-Nestes zum Operations-Nest vermeiden.

- tionen direkt auf dem zugeordneten Nest aufgerufen werden oder indirekt über das Operations-Nest in Auftrag gegeben werden<sup>3</sup>. Es wäre jedoch zu überlegen, die Operationen *lock* und *unlock* ausschliesslich mittels generischer Operationen zu realisieren und damit nicht mehr als Elementaroperationen zu betrachten, da diese für die Herstellung von generischen Operationen nicht benötigt werden; damit würden die Schnittstellen-Mechanismen aus Abschnitt 2.3.2 die Eigenschaft der Minimalität erfüllen.

Weiterhin bietet ein Operations-Nest die Möglichkeit zur *atomaren Bündelung* von Lock-Operationen. Dazu werden mehrere Lock-Operations-Beschreibungen in ein gemeinsames Datenpaket gepackt. Die Ausführung erfolgt entweder zusammen als atomare Einheit oder gar nicht. Damit lässt sich insbesondere das Handwerker-Problem [Jür73] ohne Einführung weiterer Konzepte lösen. Bei der Bündelung mehrerer *wait*-Operationen ließe sich die Semantik disjunktiven Wartens ohne Einführung zusätzlicher Konzepte realisieren.

Nun zur Problematik der Polymorphie. Aufrufer und Bearbeiter müssen eine generische Operations-Nachricht auf ~~z~~ der passende Weise interpretieren, sonst entsteht eine Fehlfunktion des Systems.

Informationen darüber aufzubewahren, wie ein Nest-Inhalt zu interpretieren ist, ist das Konzept des Meta-Nestes vorgesehen (vgl. Operation *get\_meta* in Abschnitt 3.4.5; vgl. Abschnitt 4.2)<sup>4</sup>. Dies kann nun analog auf Operations-Nester übertragen werden. Das Operations-Meta-Nest soll einen *Konsens* zwischen dem Aufrufer und dem Bearbeiter über die zu verwendenden *Datenformate* bzw. *Datentypen* vermitteln.

Wir brauchen also ein *Typsystem*, wie es prinzipiell bei Programmiersprachen schon lange eingesetzt wird. In der Literatur dieses Gebiets wenden unzählige Varianten von Typsystemen behandelt. Die Auswahl eines konkreten Typsystems fällt leichter, wenn man sich folgende Anforderungen vor Augen hält, die auch schon bisher in Betriebssystemen traditionell eine große Rolle gespielt haben:

1. Eine Typprüfung (d.h. ein Test, ob zwei Datenformate ineinander passen) *braucht nur* zur Laufzeit erfolgen.

Es gibt genügend Anwendungsfälle, wo dies auf jeden Fall erst zur Laufzeit geschehen kann; dynamische Typprüfungen sind also unbedingt erforderlich und können statische Typprüfungen notfalls vollständig ersetzen<sup>5</sup>. Als Konsequenz hieraus brauchen statische Typprüfungen höchstens bei einfachen Grund-Datentypen und nicht notwendigerweise bei komple-

<sup>3</sup>In letzterem Fall ist ein *asynchroner Aufruf* möglich. Wie viel Sinn dies macht, sei dahingestellt, da die Elementaroperationen eigens zur Darstellung von Asynchronität entwickelt wurden.

<sup>4</sup>Das beim RPC oft als notwendig betrachtete Marshaling (Umwandlung zwischen inkompatiblen Datenformaten) wird hier nicht in die Grundmechanismen hineingesteckt, sondern kann bei Bedarf durch Anpassungsbausteine implementiert werden, die verschiedene Datenformate ineinander umwandeln.

<sup>5</sup>Bei genauer Betrachtung aktueller monolithischer Kern-Implementierungen im Unix-Umfeld fällt auf, dass diese exzessiven Gebrauch von dynamischen Typprüfungen machen. Dies wird z.B. durch die Existenz der Fehler-Rückmeldung EINVAL (Invalid Argument) in Unix belegt. Es gibt kaum Systemaufrufe, die Parameter verlangen, wo statt der Bearbeitung eines Aufruf-Auftrages nicht dieser Fehlercode als Rückmeldung zurückkommen kann.

ten Datentypen vorgesehen zu werden. Dies vereinfacht ein Typsystem *enorm*.

2. Welche Typen (d.h. Datenformate) ein Bearbeiter akzeptieren und bearbeiten kann, kann nur dieser alleine festlegen. Typ-Informationen dienen zur Einweg-Kommunikation zwischen Bearbeiter und Aufrufer; damit kommen komplexere Mechanismen wie Typ-Inferenzsysteme nicht *auf Systemebene*<sup>6</sup> in Frage.

Es ist klar, dass trotz dieser Einschränkungen viele verschiedene Typsysteme und noch mehr konkrete Realisierungen derselben in Frage kommen. In nächsten Abschnitt werde ich zwei einfache Typsystem vorstellen, die wichtige Grundfunktionen abdecken sollten und lediglich als Beispiele zu verstehen sind.

## 6.1 Beispiel-Typsysteme

Einige Betriebssystem-Konstrukteure haben sich in der Vergangenheit geweigert, Typkonzepte einzuführen, was u.a. zu der historischen Spaltung zwischen Betriebssystem- und Datenbank-Konstrukteure beigetragen hat. Argumente hierfür waren oder sind immer noch, dass im Betriebssystem fest verankerte Typen die Flexibilität mindern, die Schnittstellen verkomplizieren und obendrein Performanz kosten. Ich hoffe, diese Sichtweise relativieren zu können. Es geht mir nicht darum, fest verankerte Typen einzuführen, sondern *auf der Basis generischer Schnittstellen* die Deklaration *beliebiger* Typen zu ermöglichen, indem es *erweiterbar* ist. Typbeschreibungen stellen ein *optionales* Konzept dar, das auch weggelassen werden kann; dann erhält man Generizität in Reinkultur. Richtig eingesetzt, kann es generische Schnittstellen ergänzen und vereinfachen, indem traditionell von Hand codierte Vorgänge (beispielsweise bei Verwendung von Assembler bis in 1970er Jahre hinein) *automatisiert* werden. Ich versuche in den folgenden Beispielen zu zeigen, dass exzellente Performanz von Typprüfungen durch die Wahl geeigneter Sprachklassen und Kodierungen erreichbar ist.

### 6.1.1 Minimal-Typsystem

Hier geht es um die Frage, was ein Typsystem *unbedingt* enthalten muss, damit generische Operationen einigermaßen brauchbar und benutzbar sind. Alles, was dazu nicht erforderlich ist, versuchen wir weg zu lassen und auf Laufzeit-Tests durch den Bearbeiter zu verlagern.

Wenn ein Operations-Nest mehrere verschiedene generische Operationen ermöglichen soll, dann gibt es eine Information, die in jedem Falle sowohl beim Aufrufer als auch beim Bearbeiter vorhanden sein muss: der *Name* der Operation.

Eine einfache Realisierung eines Minimal-Typsysteems besteht darin, dass das Meta-Nest des Operations-Nestes

<sup>6</sup>Damit sind Typ-Inferenzsysteme jedoch nicht generell ausgeschlossen: die Abschnitt 4.2.2 vorgestellten *strategy\_\**-Bausteine können nicht nur Laufzeit-Test auf Typkompatibilität auszuführen, sondern im Prinzip beliebige Folge-Instantierungen mit beliebigen Parametern auslösen. Damit sollte es theoretisch möglich sein, sogar ein Hindley-Milner-Typsystem zu realisieren.

eine Liste der implementierten zueinander disjunkten Operationsnamen enthält. Diese Liste lässt sich beispielsweise als lückenlos liegende Folge von Paketen realisieren, deren Grenzen im adjungierten Nest des Meta-Nestes angezeigt werden. Für die Kodierung der Operationsnamen kommen beispielsweise Text-Zeichenketten in Frage, die gegenüber Integer-Konstanten<sup>7</sup> zu bevorzugen sind.

Mehr braucht ein minimales Typsystem nicht zu enthalten<sup>8</sup>; ob die Anzahl, Größen und Werte der Operations-Parameter korrekt sind, kann der Bearbeiter selbst prüfen und ggf. mit der Rücklieferung einer Fehlermeldung reagieren.

### 6.1.2 Einfaches erweiterbares Typsystem

Das soeben vorgestellte minimale Typsystem ist bereits insofern *erweiterbar*, als dass der Wertevorrat für die Kodierung der Operationsnamen so groß gewählt werden kann, dass zukünftige Erweiterungen um neue Operationen praktisch unbeschränkt möglich sind. Es geht jetzt um die Frage von *Schnittstellen-Erweiterungen*, die man unter die Rubrik „Erweiterungs-Generizität“ (vgl. Abschnitt 2.1.2) einordnen kann<sup>9</sup>. Bei der hier vorgestellten Architektur ist vor dem Einsatz von Erweiterungs-Generizität unbedingt gewissenhaft zu prüfen, ob der gleiche Effekt nicht auch durch kompositorische Generizität (vgl. Abschnitt 2.1.3) erreichbar ist. Kompositorische Generizität hat grundsätzlich Vorrang vor Erweiterungs-Generizität; dieses Philosophie ist ein wichtiges Unterscheidungsmerkmal des hier vorgestellten Ansatzes gegenüber den meisten konkurrierenden Ansätzen.

In der Literatur aus dem Gebiet der Programmiersprachen werden Typsysteme als *algebraische Struktur* aufgefasst, für die eine *abstrakte Syntax* angegeben werden kann und meist auch nur angegeben wird. Dies genügt i.A. für unsere Zwecke nicht: ein Betriebssystem muss auch die *konkreten Datenformate* (sog. Aufruf-Konventionen) festlegen, da die System-Schnittstellen übergreifend für unterschiedliche Aufrufer-Typen gelten sollen, insbesondere den Anschluss an unterschiedliche Programmiersprachen und Laufzeitsystem-Modelle ermöglichen sollen. Wir benötigen daher eine Spezifikation der algebraischen Struktur in *konkreter Syntax*<sup>10</sup>.

<sup>7</sup> Diese werden z.B. in Unix zur Unterscheidung der durchzuführenden Operationen bei `ioctl` und `fcntl` eingesetzt. Über die damit verbundenen Probleme und Auswirkungen ist bereits genügend in der Folklore-Literatur diskutiert worden. Bei der Geschwindigkeit heutiger Prozessoren kann der minimal größere Aufwand beim Zugriff auf Zeichenketten gegenüber dem Zugriff auf ein Maschinenwort kein schlagendes Argument mehr sein, vor allem, wenn man den Zugewinn an Komfort dagegen stellt.

<sup>8</sup> Man könnte theoretisch sogar noch die Liste der Operationsnamen weglassen. Der Bearbeiter kann dann zwar das Datenformat aller eingehenden Aufträge selbst prüfen, der Aufrufer ist in diesem Falle aber vollkommen „blind“ und kann die Operations-Namen nur erraten, wenn er sie nicht aus einer anderen externen Quelle (wie z.B. eine Schnittstellen-Definition) weiß. Wenn überhaupt keine Typinformation kommuniziert wird, dann handelt es sich nicht um ein Typsystem im eigentlichen Sinne.

<sup>9</sup> Diese Problematik taucht in Betriebssystemen immer wieder auf, und zwar nicht nur in solchen, die Objektorientierung als Leitmotiv angeben. Microsoft hat beispielsweise den Begriff der „DLL-Hölle“ geprägt. Damit sind zueinander inkompatible Versionen von Systembibliotheken gemeint, die aufgrund intern geänderter Schnittstellen nicht zusammenpassen.

<sup>10</sup> Beispiele für die konkrete Syntax von system-übergreifenden Schnitt-

Zur Spezifikation einer konkreten Syntax eignen sich bekanntermaßen *kontextfreie Grammatiken*. Diese bringen jedoch in ihrer Allgemeinform das Problem der *Mehrdeutigkeit* mit sich: bei unbedachter Verwendung zur Beschreibung von Datenformaten kann es vorkommen, dass ein konkret vorliegendes Datenformats-Muster verschiedene Interpretationen zulässt. Aus der Theorie der formalen Sprachen ist bekannt, dass die Mitgliedschaft einer beliebigen Grammatik in der Klasse der *eindeutig kontextfreien Sprachen* nicht entscheidbar ist; zum Glück gibt es jedoch entscheidbare Unterklassen wie die bekannten *LR(k)* oder *LL(k)*-Grammatikklassen. Diese sind nicht nur entscheidbar, sondern reichen für praktische Zwecke vollkommen aus<sup>11</sup>.

Ich möchte noch einen Schritt weitergehen und zeigen, dass für die Zwecke von Betriebssystemen die Klasse derjenigen *LL(1)*-Grammatiken ausreichend ist, die sich zusätzlich in Greibach-Normalform<sup>12</sup> befinden. Das Wortproblem bzw. das Parsing-Problem ist bei diesen Grammatiken trivial lösbar, da man beispielsweise das bekannte Verfahren des *rekursiven Abstiegs* in *interpretativer Weise* direkt auf einer Kodierung der Grammatik ausführen kann, ohne die bei *LL(1)* notwendigen FIRST-Mengen berechnen zu müssen, da diese trivialerweise mit dem ersten Terminalsymbol übereinstimmen, mit dem jede Grammatikregel wegen der Greibach-Normalform beginnen muss. Die beim rekursiven Abstieg notwendigen Parsing-Entscheidungen werden dadurch trivial.

Um sicherzustellen, dass eine Greibach-Grammatik auf jeden Fall die *LL(1)*-Eigenschaft erfüllt, gibt es eine einfache Methode, die leicht zu erlernen und zu beherrschen sein dürfte: Man verlangt als *Ersatz-Forderung* für die *LL(1)*-Eigenschaft, dass keine verschiedenen Produktionsregeln der Form  $X \rightarrow a\alpha$  und  $X \rightarrow a\beta$  mit gleichem Nichtterminalsymbol  $X$  und gleichem Terminalsymbol  $a$ , aber verschiedenen Fortsetzungen  $\alpha$  und  $\beta$  vorkommen dürfen. Diese Forderung lässt sich dadurch einhalten, dass man für  $\alpha$  bzw.  $\beta$  ein neues Nichtterminalsymbol  $Y$  einführt, das die Rolle der bisherigen Reste  $\alpha$  und  $\beta$  gemeinsam ausführen und übernehmen soll. Da alle Produktionen für  $Y$  wiederum den gleichen Erfordernissen unterliegen, ergibt sich die *LL(1)*-Eigenschaft beim *Entwurf* der kontextfreien Grammatik durch einen Menschen ganz von selbst. Ich nenne eine Greibach-Grammatik mit dieser Ersatz-Forderung eine *Typ-Grammatik*.

Eine Typ-Grammatik setzt die folgenden bekannten Grundprinzipien des Entwurfs von Datenstrukturen direkt in leicht zu handhabende Grammatik-Regeln um:

- Die Schachtelung
- Die Sequenz
- Die Alternative

stellen finden sich u.a. in den Internet-RFCs (Request For Comments). Datenformate werden dort fast durchwegs durch kontextfreie Grammatiken in einer zur Backus-Naur-Form ähnlichen Notation angegeben.

<sup>11</sup> Ein Beleg hierfür sind die formalen Spezifikationen von Netzwerk-Protokollen und -Datenformaten in den Internet-RFCs. Diese scheinen mir allesamt in der Klasse *LL(k)* zu liegen.

<sup>12</sup> Bei der Greibach-Normalform müssen alle Produktionsregeln die Form  $X \rightarrow a\alpha$  haben, wobei  $X$  ein Nichtterminalsymbol,  $a$  ein Terminalsymbol, und  $\alpha$  eine beliebige Zeichenfolge aus Terminal- oder Nichtterminalsymbolen ist.

Eine Schachtelung von Datenformaten wird durch die Unterscheidung zwischen Terminal- und Nichtterminalsymbolen ausgedrückt; ein Nichtterminalsymbol lässt sich an verschiedenen Stellen *wiederverwenden*. Eine Sequenz von Datenformaten<sup>13</sup> wird durch Hintereinander-schreiben von Terminal- oder Nichtterminalsymbolen in einer Grammatikregel ausgedrückt; mathematisch gesehen repräsentiert es ein kartesisches Produkt. Alternativen zum gleichen Nichtterminal müssen im Gegensatz zu beliebigen kontextfreien Grammatikregeln *stets* durch voneinander verschiedene Terminal-symbole<sup>14</sup> eingeleitet werden, so dass die jeweils richtige Alternative stets am Anfang eines rekursiven Abstiegs durch einen *trivialen Diskriminat*or<sup>15</sup> erkannt werden kann.

Im Gegensatz zu Programmiersprachen wird diese Methodik in Betriebssystemen vollkommen dynamisch eingesetzt; die Erkennung eines Datenformats kann i.A. nicht vom Typsystem eines Compilers vorweggenommen werden, sondern muss mindestens bei der Verdrahtungs-Operation zur Laufzeit durchführbar sein. Hierfür eignet sich die *Laufzeit-Interpretation*<sup>16</sup> von Datenformaten. Kontextfreie Grammatikregeln ganz besonders. Grammatiken in deterministische Kellerautomaten sind dadurch nicht ausgeschlossen<sup>17</sup>.

Eine konkrete Realisierung mittels interpretierbaren Typ-Grammatiken könnte beispielsweise folgendermaßen aussehen:

*Nichtterminalsymbole* werden durch ASCII-Zeichenketten aus Großbuchstaben und Unterstrich dargestellt. Ein Beispiel wäre „IP\_ADDRESS“.

*Terminalsymbolklassen* verhalten sich logisch wie Nichtterminale, zu denen keine Regeln definiert werden dürfen, da es bereits fest eingebaute Regeln für sie gibt<sup>18</sup>. Sie werden durch ASCII-Zeichenketten in Kleinschreibung dargestellt, denen eine ASCII-kodierte Zahl angefügt ist, die den Platzbedarf in Bytes beschreibt. Beispiele sind „int1“, „int2“, „int4“ oder „int8“, die vorzeichen-behaftet zu interpretierende Maschinenwörter der jeweiligen Längen sollen, oder „space512“ für einen uninteressanten Datenblock. Weitere Terminalsymbolklassen wie „Indian\_unsigned\_int4“ lassen sich später jederzeit hinzufügen; welche Bedeutung sich hinter dem Namen verbirgt, ist für das Parsen des Datenformats egal, nur der Platzbedarf ist entscheidend.

*Terminalsymbole* werden durch Hintereinander-schreiben einer Terminalsymbolklasse und einer kon-

<sup>13</sup> Typische Vertreter von Sequenzen sind Arrays und Records.

<sup>14</sup> Bei allgemeinen kontextfreien Grammatiken können Alternativen Regeln mehrere logische Alternativen repräsentieren, die logische Alternativen zu verschiedenen Themen darstellen und nichts im logischen Sinne miteinander zu tun haben brauchen.

<sup>15</sup> Dieses Konzept taucht z.B. in den Tags von varianten Records von Pascal-ähnlichen Programmiersprachen auf; in Ada ist die Verwendung eines Tag-Feldes sogar Pflicht.

<sup>16</sup> Dies schließt nicht die Verwendung eines Prü-Compilers aus, der immer wiederkehrende Teile der Interpretation einmal vorweg berechnet (sog. *Precomputing*).

<sup>17</sup> Ob die Vorübersetzung in ausführbare Maschinenbefehle („just-in-time compiler“) weitere Vorteile bringen kann, wäre zu untersuchen.

<sup>18</sup> Dieses Konzept taucht in Compilern als Token-Klasse auf, z.B. die Token-Klasse aller Identifier oder aller Zeichenketten.

kreten Ausprägung in Klammern notiert. Beispiele sind „int2(7)“ oder „int4(-1)“. Für Zeichenketten variabler Länge kann man eine Konvention der Art „string('Name')“ einführen, bei der sich der aktuelle Platzbedarf aus dem Text zwischen den Hochkommata ergibt.

Grammatikregeln lassen sich beispielsweise auf folgende Weise kodieren:

„STRUCTURE=int1(0); SUBSTRUCTURE; int4;“ Wie zu erkennen ist, werden die Strichpunkte nicht nur als Trennzeichen, sondern auch als Abschlusszeichen einer Regel eingesetzt. Die Typgrammatik-Eigenschaft ist erfüllt, wenn keine zwei Regeln vorhanden sind, die bis zum Auftreten des ersten Strichpunktes miteinander übereinstimmen; weiterhin müssen alle Regeln zum gleichen Nichtterminal „STRUCTURE“ hinter dem Gleichheitszeichen mit der gleichen Terminalsymbolklasse beginnen. Falls ein Nichtterminal keine Alternativen enthalten soll oder darf, dann ist als Sonderfall eines Terminalsymbols auch „0(0)“ zulässig, das einen Diskriminatator repräsentiert, der keinen Platz beansprucht und von dem es deshalb nur eine einzige Ausprägung gibt (so dass als Folge davon keine weiteren Alternativen mehr zulässig sind).

Für das *Startsymbol* der Grammatik kann man eine Konvention einführen, beispielsweise ein Nichtterminalsymbol „OP“ für die Beschreibung von Operations-Nestern, oder „DATA“ für die Beschreibung von Datenformaten in persistenten Nestern. Die Beschreibung der vier Grundoperationen sieht dann beispielsweise folgendermaßen aus:

```
OP=string('get'); int4; addr8; int8; addr8;
int1; res_addr8; res_int8; res_int1;
```

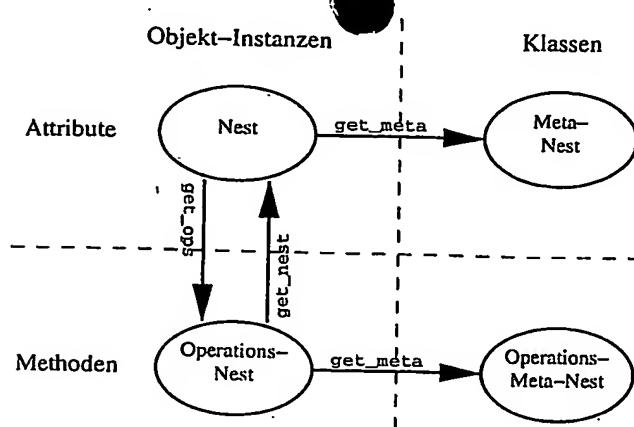
```
OP=string('transfer'); int4; addr8; int1;
int1; addr8; res_int1;
```

```
OP=string('wait'); int4; addr8; res_int1;
```

```
OP=string('put'); int4; addr8; int1;
```

Dieses Beispiel stellt nur die wichtigsten Grundelemente eines Typsystems vor und sollte für den praktischen Einsatz um weitere Ausdrucksmöglichkeiten erweitert werden, etwa Feldnamen zur Beschreibung von Records oder weitere Terminalklassen zur Beschreibung von Bitfeldern oder von Bereichstypen, sowie von Ausrichtungs-Konventionen („alignment“). Mit Hilfe weiterer Konventionen ist beispielsweise die automatisierte Überprüfung der Gültigkeit von Adressbereichen oder von Deskriptoren möglich, so dass diese nicht mehr wie in bisherigen Betriebssystem-Implementierungen „von Hand“ ad hoc getestet werden muss.

## 6.2 Zusammenhang mit Objektorientierung

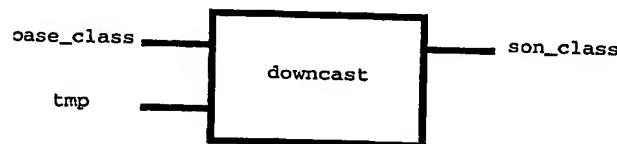


Das Bild zeigt den Zusammenhang der vier vorgestellten Nest-Arten mit bekannten Konzepten der Objektorientierung. Die gestrichelten Trennlinien sollen eine Unterteilung des Feldes nach zwei verschiedenen Kriterien andeuten:

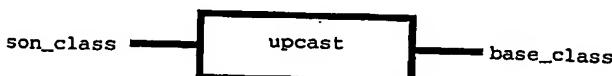
- Unterteilung in Objekt-Instanzen und Klassen: Nest-Instanz versus zugehöriges Meta-Nest
- Unterteilung in Attribute und Methoden: Daten-Nest versus Operations-Nest

Die Beziehungen von Instanzen der jeweiligen Nest-Art sind durch Pfeile mit Beschriftung der zugehörigen Zugriffsfunktionen gekennzeichnet. Eine Nest-Instanz steht mit ihrer zugehöriger Operations-Nest-Instanz in einer 1:1-Beziehung, genau wie beim Zusammenpacken von Attributen und Methoden in der Objektorientierung zu einer Objekt-Instanz. Der *Status* einer Objekt-Instanz wird dabei in der Nest-Instanz abgespeichert.

Der Zusammenhang mit der *Vererbung* lässt sich durch zwei Anpassungs-Bausteine darstellen, die entsprechende Transformationen auf den Nestern bzw. Meta-Nestern durchführen.



Beim *downcast* wird die Schnittstelle erweitert; zu den am Eingang *base\_class* verfügbaren Attributen und Methoden können neue hinzukommen, die am Ausgang *son\_class* zur Verfügung gestellt werden; ggf. kann dabei auch die *Implementierung* (bzw. das *Verhalten*) einiger Methoden abgeändert werden (*überschriebene Methoden* und *virtuelle Methoden*). Da sich dabei der Umfang des abgespeicherten Status vergrößern kann, ist ein Eingang *tmp* vorgesehen, um diesen aufzunehmen. Bei geeigneten Konventionen über die Platz-Allokation im *base\_class*-Nest lässt sich der Status aber auch dort abspeichern, so dass *tmp* überflüssig wird.

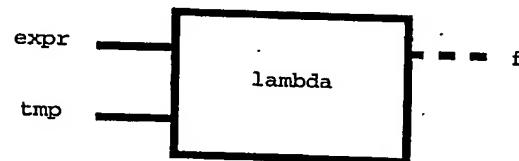


Beim *upcast* wird lediglich die Schnittstelle verkleinert, so dass nicht mehr alle Attribute und Methoden von außen<sup>19</sup> zugreifbar sind (Prinzip der *Verbergung*).

Aus dieser Beschreibung wird ersichtlich, dass die Konzepte der Objektorientierung einen *Spezialfall* der hier vorgestellten Methodik darstellen. Durch die Abstraktionen Nest und Baustein sind folgende Konzepte realisierbar, die bei *üblichen*<sup>20</sup> Ausprägungen von Objektorientierung im Regelfall nicht vorgesehen sind:

- Austausch einzelner Methoden-Implementierungen zur *Laufzeit*, und zwar unabhängig von einer Klassenhierarchie<sup>21</sup>
- Änderung der *Repräsentation* von Attributen (beim Erweitern der Schnittstelle und/oder zur Laufzeit); vgl. Anhang B.
- Aufspaltung eines Status-Raums in beliebige Kombinationen von Teilräumen<sup>22</sup>
- Einfache Verwendung *beliebig unterschiedlicher* Methoden-Sätze<sup>23</sup>

### 6.3 Zusammenhang mit dem Funktionalen Paradigma



Die Grundidee eines *lambda*-Bausteins ist, einen *universellen Interpreter* zu schaffen, dessen tatsächliche Funktionsweise durch den Inhalt des *expr*-Eingang in einer geeigneten Kodierung festgelegt wird. Am Ausgang *f* werden

<sup>19</sup> Eine Verkleinerung lässt sich dadurch erzielen, dass Attribute und Methoden in den jeweiligen Meta-Nestern nicht mehr erscheinen, so dass sie aus dem Typsystem ausgeblendet erscheinen. Ein besserer Zugriffsschutz lässt sich dadurch realisieren, dass bei der Transformation auch Ausblendungen in den jeweiligen adjungierten Nestern vorgenommen werden.

<sup>20</sup> Eine Ausnahme ist Smalltalk, wo Methoden-Änderungen oder sogar Klassen-Änderungen zur Laufzeit prinzipiell möglich sind, da alle Strukturen dynamisch angelegt sind.

<sup>21</sup> Beispiele sind Pipe-Operationen in Linux, deren Semantik vom Modus abhängt, in dem die Pipe geöffnet wurde. Es gibt Implementierungen, die jeden Modus als eigene Prozedur realisieren und über eine einheitliche generische Schnittstelle verfügbar machen.

<sup>22</sup> Übliche Schutzkonzepte von objektorientierten Programmiersprachen wie C++ oder Java sehen eine global gültige Unterteilung in private und öffentliche Regionen vor. Demgegenüber erlaubt das Konzept der adjungierten Nester auch Sichten, die für jede Teilnehmer-Instanz anders ausfallen können.

<sup>23</sup> Bei der objektorientierten Vererbung werden Methoden-Schnittstellen stets nur *erweitert*. Bausteine lassen hingegen auch das *Verdecken* bisheriger Attribute oder Methoden zu. Ich halte dies für sehr nützlich, weil es das *Steigern der Abstraktionsstufe* in einer Baustein-Hierarchie ermöglicht: logisch betrachtet handelt es sich nach einer Steigerung der Abstraktionsstufe zwar immer noch um dieselbe Objekt-Instanz, die Zugriffsmethoden werden jedoch „abstrakter“. Man kann Benutzern der höheren Abstraktionsstufen untersagen, diese mit Methoden niedrigerer Stufe gemischt zu verwenden, um beispielsweise die Umgehung von erst weiter oben gegebenen *Zusicherungen* zu unterbinden („schwarze Schnittstellen“).

besondere generische Operationen zur Verfügung gestellt, die über die gestrichelt gezeichneten Leitungen vom nachfolgend vorgestellten `apply` verwendet werden können:



Eine von `lambda` erzeugte Funktion stellt keine vollständige Space-Instanz mit einem Zustand dar, sondern stellt lediglich die Operationen bereit, die die Funktion `f` realisieren, wenn sie mit Hilfe von `apply` und einem Argument `x` zu einer funktionsfähigen Nest-Instanz `f(x)` verknüpft werden. Da dabei zusätzlich zum in `x` gehaltenen Status weiterer Platzbedarf für Status-Information zur Berechnung von `f(x)` erforderlich sein kann, wird wie für solche Fälle üblich ein `tmp`-Eingang vorgesehen.

Die Funktionsweise ist wie folgt skizziert: wenn am Ausgang `f(x)` irgendwelche Nest-Operationen aufgerufen werden, dann ruft `apply` die generischen Operationen des `f`-Eingangs mit den passenden Daten-Argumenten auf, die es vorher aus `x` gewonnen hat (und übergibt ggf. weiteren aus `tmp` gewonnenen Hilfsspeicher, der Instanz-spezifisch ist).

Funktionen höherer Ordnung [Fie88] lassen sich nach dem gleichen Prinzip erzeugen. Am Ausgang von `apply` erscheinen dabei lediglich keine voll funktionsfähigen Nester, sondern nur Operations-Nester, die mit Hilfe weiterer `apply`-Instanzen erst normal benutzbare Nester ergeben.

Bausteine und ihre Verdrahtungen sind für sich genommen bereits universelle Konstrukte, die sich beispielsweise auch mit theoretisch interessanten Beschreibungen wie den Hotz'schen X-Kategorien beschreiben lassen.

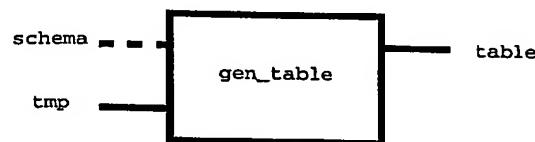
Weiterhin ist zu erwähnen, dass man rein theoretisch bekannte Verfahren zur Term- oder Graph-Reduktion auch durch Bausteine in den Steuerleitungen von `con` (Abschnitt 4.2.1) realisieren könnte. Theoretisch könnte man damit auch rein syntaktische Evaluationen mit Hilfe in den Baustein-Verdrahtungen gespeicherten Beziehungen betreiben, ohne den Zustand der beteiligten Nester benutzen zu müssen. Die Baustein-Verdrahtungen werden dabei zur Repräsentation einer algebraischen Struktur missbraucht. Dieser theoretische Zusammenhang beleuchtet jedoch die Wichtigkeit und Notwendigkeit, in der Praxis sehr auf die in den Verdrahtungen gespeicherte Information und auf ihre nicht immer leicht zu beherrschenden Konsequenzen zu achten.

Obwohl man in der Praxis den hier aufgezeigten Zusammenhang mit funktionalen Paradigmen eher nicht in der theoretischen Form des allgemeinen Lambda-Kalküls einsetzen sollte (da Nester nicht als Konkurrenz zu etablierten Laufzeitsystemen für funktionale Berechnungen konstruiert wurden), dürften speziellere Formen funktionaler Denk- und Konstruktionsmethoden durchaus wertvolle praktische Anwendungen haben: Universell einsetzbare Interpreter brauchen nicht unbedingt pure funktionale Para-

digmen zu beachten, sondern können in der Praxis durchaus „hemdsärmelig“ wie beispielsweise durch einen Perl-Interpreter [Haj00] realisiert werden, der seinen Programmcode aus einem Eingang bezieht, der von den zu bearbeitenden Daten getrennt ist<sup>24</sup>. Damit lassen sich beispielsweise Prototypen sehr schnell entwickeln.

## 6.4 Zusammenhang mit Datenbank-Schemata

Relationale Datenbank-Schemata lassen sich mittels Nester und Bausteinen erzeugen, umwandeln, in Datenbank-Instanzen umsetzen, und die instantiierten Daten verarbeiten.



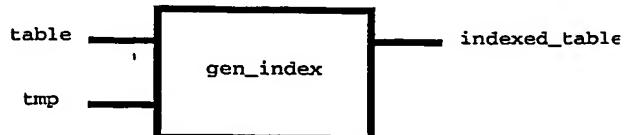
In einem Nest lassen sich Datenbank-Tabellen und Indizes ablegen, das zugehörige (Teil-)Schema wird im Meta-Space bzw. in den Attributen in einer geeigneten Repräsentation gehalten. Durch einen Baustein `gen_table` kann ein reines Schema, das ein oder mehrere Metabeschreibungen von Tabellen enthält, in ein oder mehrere<sup>25</sup> entsprechende Tabellen instantiiert werden. Der persistente Status wird im `tmp`-Eingang gehalten und am Ausgang zur Verfügung gestellt. Es wird dafür gesorgt und geprüft, dass die instantiierten Datensätze sowohl syntaktisch als auch semantisch dem Schema-Aufbau entsprechen.

Datensätze fester Länge lassen sich am einfachsten als zusammenhängendes Array abspeichern. In diesem Fall braucht das adjungierte Nest nicht unbedingt die Grenzen jedes einzelnen Datensatzes darzustellen, da diese Information leicht berechnet werden kann. Bei variabler Datensatz-Länge ist allerdings die Repräsentation lückenlos liegender Pakete (vgl. Abschnitt 3.4.1) im adjungierten Nest angebracht. Falls aus Gründen der Uniformität diese Art der Darstellung auch bei festen Datensatzlängen gefordert wird, lässt sich der Inhalt des adjungierten Nests intern auf virtuelle Weise berechnen, ohne Platz im persistenten Speicher zu verbrauchen. Eine weitere Möglichkeit besteht darin, die Bestandteile von Datensätzen mit fester Länge von denjenigen mit variabler Länge zu trennen, so dass eine performante Array-Darstellung der Datensätze mit Verweisen auf einen Heap-Bereich entsteht, der die variablen Teile beherbergt; derartige Verweise entsprechen konzeptionell den Datensatz-Beschreibungen im adjungierten Nest und sollten aus Gründen der Uniformität an einer Stelle zentralisiert werden; das Konzept des adjungierten Nests eignet

<sup>24</sup>Ein klassisches Beispiel universeller Interpreter-Konstrukte höherer Ordnung ist der `#!`-Operator zusammen mit der Argumentübergabe an Prozesse in Unix, mit dem sich beliebige Interpreter starten lassen, die ihrerseits wieder beliebige weitere Operationen auf beliebigen Daten ausführen können. Eine Nachbildung durch Bausteine ist leicht möglich.

<sup>25</sup>Mehrere Tabellen, Indizes etc. sollten sich durch Vorreservierung genügend grosser Adressbereiche aus mindestens 64 Bit Adressraum-Gesamtgröße herauslösen lassen, so dass Verschiebungen ganzer Tabellen gar nicht oder nur extrem selten benötigt werden.

sich hierfür besonders<sup>26</sup>. Um eine gute Kombinierbarkeit von Bausteinen zu erhalten, sollten diese Konventionen einheitlich festgelegt werden.



Über die Sortierung von Datensätzen wurde bisher nichts gesagt. Es ist möglich, eine Tabelle gleich von vornherein nach einem Primärschlüssel sortiert zu halten. Da Sekundärschlüssel jedoch unabhängige Sichten erlauben, ist es günstig, die Schlüssel unabhängig von ihrer Art und Anzahl ebenfalls im Nest zu halten. Eine Schlüssel-Repräsentation kann beispielsweise als Array von Index-Nummern der Datensätze erfolgen, die nach dem Schlüssel sortiert sind. Einfügungen und Löschungen von Datensätzen werden dann durch move-Operationen auf den jeweiligen Arrays bewerkstelligt. Natürlich ist es ebenfalls möglich, klassische Repräsentationen wie B<sup>+</sup>-Bäume zu verwenden, doch sind diese wesentlich komplizierter aufgebaut und schwerer zu interpretieren. Da Nester vorzugsweise universelle Generizität mittels sehr einfacher Datenstrukturen repräsentieren sollten, halte ich die Verwendung sortierter Arrays für günstiger, zumal eine billige move-Operation zur Verwaltung dieser Array-Strukturen zur Verfügung steht. Performante Suchstrukturen wie B-Bäume werden dadurch nicht überflüssig; sie werden lediglich an anderer Stelle einer Baustein-Hierarchie eingesetzt, nämlich bei der Erzeugung dynamischer Nester (Baustein map in Abschnitt 4.1.3).



Die Operationen der Relationen-Algebra wie Selektion, Projektion und Kartesische Produktbildung durch verschiedene join-Varianten lassen sich direkt durch Bausteine darstellen und ausführen. Details werden weiteren Arbeiten zur Integration von Datenbanken in Betriebssysteme überlassen.

<sup>26</sup>Eventuell wäre es eine gute Idee, adjungierte Nester zum adjungierten Nest einzuführen. Das adjungiert-adjungierte Nest enthält dann die Beschreibung ganzer Tabellen und Indizes, das einfach adjungierte Nest die konkrete Beschreibung der jeweiligen Tabelle.

# Kapitel 7

## Multiversion-Modelle

In diesem Kapitel soll untersucht werden, wie sich die Funktionalität von Transaktionen auf einfache generische Weise in die Nest-Schnittstelle integrieren lässt. Ziel ist nicht die vollständige Abdeckung aller nur denkbarer multiversion-Modelle, sondern die exemplarische Auswahl und Darstellung einiger (hoffentlich) für praktische Zwecke geeigneter Modelle, die hinsichtlich der gängigen Transaktions-Semantiken universelle Generizität bereitstellen sollten.

Historisch betrachtet hatten Transaktionen [BHG87, LSC91, GR93, Elmbe] das Ziel, trotz paralleler Aktivitäten jedem Konsumenten ein *singleuser*-Modell bereit zu stellen, so dass sich Programmierer nicht um Parallelität zu kümmern brauchten. Nach ausführlicher Untersuchung von Synchronisation und Rücksetzen (recovery) wurde entdeckt, dass man trotz *singleuser*-Ausgangsbasis mit *Versionen* von Datenobjekten zu tun hatte; ohne Versionierung ist Rücksetzen nicht denkbar. Daraufhin wurden Multiversions-Varianten von Transaktionen [VGH93] untersucht, insbesondere Zeitstempel-Verfahren auf Basis von Versionen (time stamp ordering, time domain addressing). Im Folgenden setze ich die Kenntnis der wesentlichen Grundkonzepte aus der Datenbank-Welt aus der genannten Literatur beim Leser voraus.

Für unsere Zwecke sind generische Transaktions-Mechanismen an der Nest-Schnittstelle gefragt, mit denen möglichst universell verschiedene Transaktions-Modelle möglich durch Bausteine implementierbar sind.

### 7.1 Anforderungen

In Anwendungsbereichen wie z.B. Finanzbuchhaltung gibt es gesetzliche Anforderungen, die das spätere Nachvollziehen (Lesen) aller durchgeführten Aktionen (Operationen) in der zeitlichen Reihenfolge der Durchführung erzwingen (lückenlose zeitliche Dokumentation aller Operationen). Darüber hinaus existieren davon unabhängige sachliche Kriterien als zeitliche Ordnungsmerkmale, unter denen durchgeführte Operationen ebenfalls nachträglich zugreifbar sein müssen. Ein typisches Beispiel aus der Finanzbuchhaltung ist der Unterschied zwischen dem Beleg-Datum eines Belegs und dem Verbuchungs-Datum: die Reihenfolge der Verbuchung (Eintragung) von Belegen muss nicht notwendigerweise in der Reihenfolge ihrer Geltung (Beleg-Datum) erfolgen. Im Journal einer Finanzbuchhaltung müssen alle Einträge mindestens in der Reihenfolge

der Eingabe, in den Konto-Auszügen sollten sie zusätzlich in der Reihenfolge ihrer sachlichen Geltung abrufbar sein. Diese Anforderungen lassen sich auf folgende Weise verallgemeinern:

1. Jede jemals durchgeführte Operations-Instanz auf einem Nest muss anhand eines Merkmals (z.B. Zeitstempel) wieder eindeutig auffindbar sein
2. Es können mehrere von einander unabhängige Zeit-Bereiche notwendig sein

Punkt 1 bedeutet, dass ein „Elefantengedächtnis“<sup>1</sup> implementierbar sein muss, da es Anwendungen gibt, die dieses aufgrund gesetzlicher Anforderungen verlangen. Falls bei anderen Anwendungen der Overhead stört, lassen sich eingeschränkte Modelle ableiten, bei denen nicht mehr benötigte Informationen gezielt oder automatisch „vergessen“ werden.

Punkt 2 scheint auf den ersten Blick zu bedeuten, dass aufgrund gesetzlicher oder sachlicher Anforderungen ein oder mehrere Totalordnungen auf ein oder mehreren Zeit-Bereichen zu implementieren wären. Dies wäre jedoch insbesondere in verteilten Systemen sehr hinderlich. Die folgenden Überlegungen befassen sich hauptsächlich mit diesem Problem.

### 7.2 Allgemeines Modell

Eine Möglichkeit wäre die Einführung von zeitlichen Halbordnungs-Konzepten in die Nest-Schnittstelle. Eine Totalordnung würde sich dann als Spezialfall dort ergeben, wo dies aufgrund von Anforderungen notwendig wäre. Diese Möglichkeit wird hier aufgrund ihrer hohen Komplexität nicht weiter untersucht<sup>2</sup>. Stattdessen schlage ich folgende Betrachtungsweise des Problems vor:

Die Idee der Festlegung einer *zeitlichen Totalordnung* zwischen Operations-Instanzen impliziert nicht notwendigerweise, dass der *Festlegungs-Vorgang* selbst in totalgeordneter zeitlicher Reihenfolge geschehen muss. Es wird lediglich verlangt, dass das *Endergebnis* eine zeitliche Totalordnung ergeben muss; wann dieses Endergebnis jedoch berechnet wird, ist eine andere Frage.

<sup>1</sup> Bekannte Realisierungen sind z.B. die Logs von Datenbanken, die gelegentlich vollständig aufbewahrt werden und als SQL-Tabelle dargestellt werden, so dass man in ihnen wie in anderen Tabellen suchen kann.

<sup>2</sup> Konzeptuell wäre dies die Verwendung Lamportscher Uhren [Lam78] auch in persistenten Speichern als dauerhaftes Modell der Darstellung zeitlicher Relationen zwischen Operationen.

Diese Unterscheidung zwischen virtueller Zeit und realer Reihenfolge von Operations-Ausführungen könnte als *lazy evaluation* bei der Festlegung der zeitlichen Reihenfolge von Operations-Instanzen charakterisiert werden<sup>3</sup>. Solange eine Totalordnung noch nicht endgültig festgelegt worden ist, können (temporäre) Halbordnungen im Effekt vorkommen. Einmal gemachte Festlegungen lassen sich jedoch nie mehr revidieren; dies ist nichts anderes als die konsequente Anwendung des Persistenz-Gedankens auf die zeitlichen Ordnungsrelationen. Ansonsten ist Persistenz jedoch unabhängig von den erreichten Zwischen- oder Endzuständen eines Nestes.

Der Endzustand eines Nestes ist erst dann erreicht, wenn nicht nur die Daten-Inhalte und Parameter aller ausgeführten Operations-Instanzen, sondern auch die Zeitstempel der Operations-Instanzen intern fest gelegt sind. Konsumenten erhalten niemals Kenntnis von halb-geordneten Zwischenzuständen; spätestens die Abfrage zeitlicher Relationen an der Nest-Schnittstelle führt automatisch zu einer Festlegung (und damit eventuell zu einer Zustands-Änderung im strengen Sinn, selbst bei einer „simpel“ Abfrage).

Die nähere Festlegung vorher (teilweise) unbestimmter zeitlicher Relationen wird *Präzisierung* genannt; alle zeitlichen Relationen zwischen Operationen eines Nestes werden zusammen gefasst *Präzisierungs-Zustand* genannt. Präzisierungs-Zustände müssen *konsistent* sein, d.h. bei einem verteilten System dürfen zwei verschiedene Abfrager keine widersprüchlichen Antworten zum gleichen Sachverhalt bekommen. Die Menge aller ausgeführten Operationen samt deren Parametern, jedoch ohne die zeitlichen Relationen, wird *Daten-Zustand* eines Nestes genannt.

Parallelität von Operationen ist insbesondere dann möglich, wenn keine *Abfrage* zeitlicher Relationen des Präzisierungs-Zustandes erfolgt (etwa weil zeitliche Ordnungen für das zu lösende Problem keine Rolle spielen). Falls niemals eine Abfrage zeitlicher Relationen zwischen Operations-Instanzen stattfindet, dann kann ein persistenter Speicher im Extremfall lauter unkorrelierte Operations-Instanzen enthalten, deren zeitliche Reihenfolge nicht festgelegt ist (eine Festlegung *darf* in diesem Fall natürlich trotzdem erfolgen).

Ob zwei Operations-Instanzen eine zeitliche Relation zu einander besitzen, könnte mit Hilfe eines *Benutzungs-Modells* implizit fest gelegt werden. Beispielsweise könnte ein bestimmtes Benutzungs-Modell vorschreiben, dass sämtliche Operationen, die sich auf gegenseitig überlappende Adressbereiche eines Nestes beziehen, in eine Totalordnung gebracht werden müssen. Es sind jedoch Fälle denkbar, in denen selbst diese schwach erscheinende Forderung nicht erfüllt zu sein braucht (beispielsweise wenn irrelevante Operations-Durchführungen einfach ignoriert und niemals mehr betrachtet werden). Da weiterhin sehr viele verschiedene Benutzungs-Modelle denkbar sind, schlage ich vor, keine Annahmen über implizite Ab-

<sup>3</sup>Beim Time-Warp [Jef85] wird zwar ebenfalls eine virtuelle Zeit-Achse verwendet, doch schreiten die Operations-Aufführungen auf dieser Achse monoton vorwärts und werden nur beim Rollback gelegentlich *intern* zurückgesetzt. Im Unterschied dazu erlaubt das hier propagierte Modell beliebige Rückdatierungen von Operationen, die unabhängig von evtl. vorkommenden Rollbacks sind; weiterhin wird die virtuelle Zeit-Achse mit der Orts-Achse zu einer zweidimensionalen Ebene verknüpft.

hängigkeiten zwischen Operations-Instanzen in die Nest-Schnittstelle aufzunehmen, sondern jegliche zeitliche Abhängigkeiten *explizit* zu machen. Die Realisierung unterschiedlicher Benutzungs-Modelle wird dadurch zur Aufgabe konkreter Baustein-Implementierungen.

Das allgemeinste *multiversion*-Modell sieht so aus, daß zu *jeder* durchzuführenden Operations-Instanz in irgend einer Form angegeben werden muß, von welchen anderen („früheren“) Operations-Instanzen sie abhängt. Die Spezifikation der „früheren“ Operations-Instanzen braucht keine individuelle Bekanntschaft aller in Frage kommenden Operations-Instanzen voraus zu setzen, sondern kann durch verschiedene mengen- oder prädikatenorientierte Spezifikationsmechanismen erfolgen.

Aus der Vielzahl von Spezifikationsmechanismen für zeitliche Ordnungen wähle ich im Folgenden eine bestimmte aus und überlasse die Untersuchung alternativer Mechanismen weiteren Forschungsarbeiten.

### 7.3 Zeitintervall-Modell

Die Grundidee besteht darin, als Zeitbasis eine *physikalische*<sup>4</sup> Lamportsche Uhr [Lam78] mit genügender Genauigkeit zu verwenden, jedoch den einzelnen Operations-Instanzen keine festen *Zeit-Punkte*, sondern *Zeit-Intervalle* (vgl. Begriff der *Linearisierbarkeit* in [HW90]) zuzuordnen. Im Unterschied zu festgefügten Konsistenzmodellen wie der linearen Konsistenz, die Halbordnungen auch im Endergebnis zulassen, könnten Zeitstempel-Intervalle zu jedem beliebigen späteren Zeitpunkt nachträglich *verkleinert* (Spezialfall einer Präzisierung) werden: ein ursprünglicher Zeitstempel  $[t_1, t_2]$  mit  $t_1 \leq t_2$  lässt sich jederzeit in  $[t_3, t_4]$  mit  $t_1 \leq t_3, t_3 \leq t_4$  und  $t_4 \leq t_2$  umwandeln.

Eine solche Verkleinerung *muss* immer dann durchgeführt werden, wenn sich beim *Vergleich* der Zeitstempel zweier Operations-Instanzen  $op_1$  und  $op_2$ , geschrieben  $t(op_1) = [t_1, t_2]$  und  $t(op_2) = [t_3, t_4]$ , eine gegenseitige Überlappung (d.h.  $t_3 \leq t_2 \wedge t_1 \leq t_4$  oder  $t_1 \leq t_4 \wedge t_3 \leq t_2$ ) ergibt. Durch die erzwungene Verkleinerungs-Operation wird sicher gestellt, dass anschliessend entweder  $t_2 < t_3$  oder  $t_4 < t_1$  gilt<sup>5</sup>, d.h. eine Kommutation der beiden Operations-Instanzen ist anschliessend nicht mehr möglich. Solange die Zeitstempel der beiden Operations-Instanzen nicht durch eine implizit oder explizit ausgeführte Vergleichs-Operation miteinander verglichen werden, dürfen sich die Intervalle überschneiden, d.h. es braucht keine endgültige Festlegung der Reihenfolge der beiden Operationen zu erfolgen.

Für den Benutzer dieses Modells sieht alles so aus, als gäbe es eine totalgeordnete globale Zeit. Die interne Realisierung verwendet jedoch (temporäre) Halbordnungen. Die

<sup>4</sup>Es geht auch mit logischen Lamportschen Uhren, nur muss der Konsument dann die „Bedeutung“ von Zeit-Werten in der Real-Zeit auf mühsame Weise selbst bestimmen.

<sup>5</sup>Bei dieser Darstellung wird still-schweigend voraus gesetzt, daß stets eindeutige Zeitstempel in einem verteilten System generiert werden. Eine wohlbekannte Methode hierfür ist das Anhängen von Knoten-Kennzeichen an den abgelesenen Wert einer Rechner-Uhr, so daß eventuell gleichlauende Uhrzeiten, die durch zufälliges „gleichzeitiges“ Ablesen von Uhren entstehen könnten, stets zu eindeutigen Zeit-Werten führt.

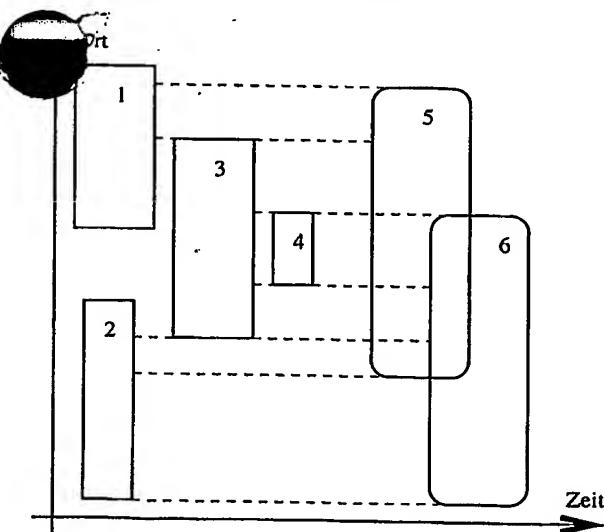
- Präsentation als Totalordnung nach aussen unterscheidet dieses Modell von explizit auf Halbordnungen aufgebauten Modellen.

## 7.4 Container-Operationen: Locks

Bereits im multiuser-Modell (Abschnitt 3.3.5) dienten Locks zur Klammerung einer Menge von Zugriffs-Operationen. In multiversion-Modellen werden Lock-Operationen analog dazu als zeitliche und räumliche *Container* aufgefasst, die eine *Menge* von Zugriffs-Operationen *beherbergen* können. Die Idee besteht darin, nur noch diese Container als Einheiten für Beziehungen zwischen Zugriffs-Operationen zu verwenden, und von den Details der im Container beherbergten Operationen und Operations-Arten zu abstrahieren.

Lock-Operations-Instanzen werden als persistente oder quasi-persistente<sup>6</sup> Objekte aufgefasst, die eine zeitliche und eine örtliche Dimension besitzen und einem Mandat als Eigentümer zugeordnet sind; verschiedene Transaktionen unter verschiedenen Mandaten laufen. Die örtliche Ausdehnung ist der Bereich des virtuellen Adressraums eines Prozesses, auf den sich der Lock bezieht. Die zeitliche Ausdehnung entspricht den soeben vorgestellten Intervallen auf einer *virtuellen Zeitachse*, die unabhängig von der tatsächlichen Ausführungs-Reihenfolge besteht. Während die örtliche Ausdehnung weitgehend festgelegt ist (nachdem sie z.B. einmal als spekulative Locks ausgehandelt wurden, vgl. Abschnitt 5.3), kann die zeitliche Ausdehnung nachträglich jederzeit verkleinert werden (Präzisierung). Ein Container darf nur solche Zugriffs-Operationen beherbergen, die sich auf einen Teil-Bereich der räumlichen und zeitlichen Dimension beziehen. Im Regelfall *erben* die von einem Container beherbergten Zugriffs-Operationen ihre zeitliche Dimension vom umgebenden Container.

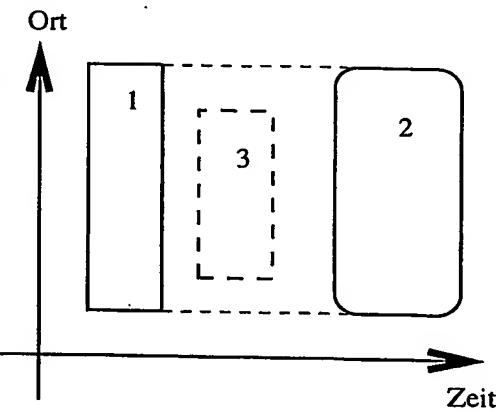
Im folgenden Bild werden Write-Lock-Instanzen durch Rechtecke mit spitzen Ecken, Read-Lock-Instanzen dagegen durch abgerundete Ecken dargestellt:



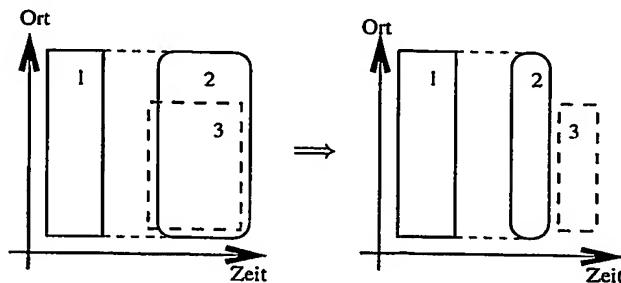
<sup>6</sup>Mit Quasi-Persistenz ist gemeint, dass grundsätzlich die Persistenz zwar angesprochen wird, jedoch später die gespeicherte Information durch gezieltes Vergessen vergröbert werden kann.

Verschiedene Write-Lock-Instanzen dürfen sich niemals gegenseitig überlappen, d.h. nicht die gleiche Fläche in der zeitlichen und örtlichen Dimension einnehmen. Im Konfliktfall lassen sich Write-Locks jedoch nachträglich immer so verkleinern, dass sie auf der Zeit-Achse nebeneinander passen (aufgrund der eindeutigen Zeitstempel, siehe Abschnitt 7.3). Read-Lock-Instanzen dürfen sich dagegen gegenseitig überlappen.

Die Verhältnisse zwischen Read- und Write-Lock-Instanzen werden im folgenden Bild näher untersucht:



Die in den Kästchen eingezeichneten Nummern sollen die Reihenfolge<sup>7</sup> andeuten, in denen die jeweiligen Locks gesetzt werden sollen. Das Setzen des gestrichelt gezeichneten Locks 3 ist nicht möglich, weil er die durch gestrichelte Linien angedeutete „Sicht“ des vorher dagewesenen Locks 2 auf Lock 1 teilweise „verdecken“ würde. Falls die Reihenfolge von Lock 2 und 3 vertauscht würde, dann wäre ein Setzen aller drei Locks möglich, jedoch würde dann die Read-Lock-Instanz eine Sicht auf die kleinere Write-Lock-Instanz erhalten. Im folgenden Sonderfall ist eine Konfliktlösung durch nachträgliches Verkleinern möglich:



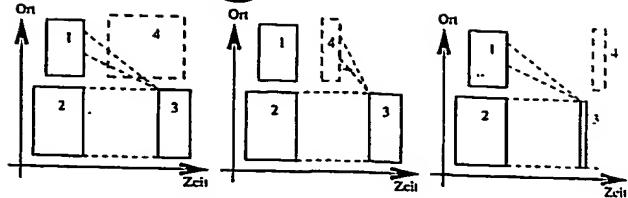
Es gilt folgender Grundsatz: Eine einmal gewährte (bzw. in Anspruch genommene<sup>8</sup>) Sicht eines Locks auf andere Locks darf nachträglich nicht verändert werden.

Die Frage ist, was man unter einer Sicht auf andere Locks verstehen soll. Intuitiv geht es um die bekannte Semantik von Speichern, nach der eine Lese-Operation bzw. ein

<sup>7</sup>Das Konzept einer „globalen Reihenfolge“ von Operations-Aussführungen braucht in einem verteilten System nicht verwendet zu werden. Die Darstellung durch Reihenfolge-Nummern dient hier nur zur Illustration.

<sup>8</sup>Es sind allgemeinere Modelle denkbar, bei denen es nicht auf die Sichten zwischen Locks, sondern zwischen den darin enthaltenen Zugriffs-Operationen ankommt. Solange im Beispiel nur Lock 2 gesetzt, aber nicht gelesen wurde, könnte man Lock 3 noch gewähren. Das sich daraus ergebende kompliziertere Modell wird in dieser Arbeit nicht untersucht.

Read-Lock den Zustand der *zeitlich letzten* davor liegenden Änderung (Write-Lock bzw. Schreib-Operationen) sehen sollte. Der Unterschied zur klassischen Semantik besteht lediglich darin, dass die Zeitachse ein virtuelles Gebilde ist, auf dem die Relationen „vorher“ und „nachher“ unabhängig von der tatsächlichen Ausführungsreihenfolge festgestellt werden (sofern eine global eindeutige Ausführungsreihenfolge überhaupt existiert).



In der Ausgangs-Situation wurde Lock 3 als kausal abhängig von Lock 1 und Lock 2 definiert. Der neu hinzukommende Lock 4 überschneidet nur den Adressbereich von Lock 1, nicht jedoch den von Lock 3. Da er nicht gleichzeitig die Adressbereiche von Lock 3 und 2 überschneidet, ist die Zulässigkeit dieser Operation eine Frage der konkreten Strategie. Ich sehe folgende Möglichkeiten:

Man kann die Ausgangssituation (erstes Bild) zulassen. Dann muss jedoch persistent gemerkt werden, von welcher Version (nämlich Lock 1) die konkrete kausale Abhängigkeit besteht. Andernfalls würde eine Verkleinerung von Lock 4 wie im zweiten Bild zu einem unbeabsichtigten Wechsel der Version führen, von der die kausale Abhängigkeit besteht. Letzteres Problem könnte auch dadurch vermieden werden, dass man unerwünschte Versionsänderungen vermeidet, indem derartige Locks nicht gesetzt werden dürfen. Die spezielle Ausgangssituation lässt hierfür zwar wegen der zeitlichen Überschneidung von Lock 4 mit Lock 3 auch die spezielle Lösung aus dem dritten Bild zu, doch führt dies im allgemeinen zu einer Behinderung.

Der einfachste Lösung scheint mir zu sein, „überspringende“ kausale Abhängigkeiten<sup>9</sup> zuzulassen, die auf den jeweiligen Lock-Instanzen und nicht auf den Adressbereichen aufgebaut sind. Überspringen ist dann zugelassen, wenn sich beide Adressbereiche von Quelle und Ziel der direkten kausalen Abhängigkeit nicht mit dem Adressbereich der eingeschobenen Lock-Instanz überschneiden. Es können jedoch weitere Zusatzbedingungen für die Zulässigkeit solcher „überspringender“ kausaler Abhängigkeiten als besondere Strategie eingeführt werden; hierzu zählt insbesondere die Einhaltung der Serialisierbarkeit von Transaktionen.

## 7.5 Kausale Abhängigkeiten

Die Verallgemeinerung der Idee einer Sicht einer Lock-Instanz auf andere Lock-Instanzen wird *direkte kausale Abhängigkeit* genannt. Die Verallgemeinerung besteht darin, dass nicht nur Abhängigkeiten wegen der (möglichen) Sicht auf andere Lock-Instanzen gleicher Adressbereiche als kausale Abhängigkeit gewertet werden, sondern darüber hinaus noch beliebige weitere kausale Abhängigkeiten in das (quasi-)persistente Präzisierungs-Zustands-Modell auf explizite Anforderung aufgenommen werden können.

Eine typische Anwendung für die Aufnahme extern spezifizierter kausaler Abhängigkeiten ist die Herbrand-Semantik klassischer Transaktionen [VGH93]. Klassische „an der Syntax“ orientierte Transaktionen betrachten jede Schreiboperation als kausal abhängig von *allen* vorher durch die selbe Transaktion gelesenen Objekten. Sogenannte „semantische“ Transaktions-Modelle [Elmbe] versuchen dagegen, durch Kenntnisse über den inneren Aufbau der Transaktionen nur diejenigen kausalen Abhängigkeiten als Konsistenz-Kriterium in Betracht zu ziehen, die auch tatsächlich vorhanden sind und benutzt werden.

Wir stellen universelle Generizität bezüglich verschiedener Transaktions-Modelle dadurch her, dass wir die explizite Angabe kausaler Abhängigkeiten in der Nest-Schnittstelle verlangen. Der Sonderfall vollständiger kausaler Abhängigkeiten von allen je von einer Transaktion angefassten Objekten lässt sich relativ leicht als Strategie durch einen geeigneten Anpassungs-Baustein implementieren.

Konkret lässt sich dies so gestalten: Read-Lock-Instanzen erhalten automatisch eine direkte kausale Abhängigkeit von allen Write-Lock-Instanzen, die sich mit ihrem Ausschnitt aus dem Adressbereich überschneiden und die in der Zeitachse davor am nächsten liegen.

Bei Write-Lock-Instanzen muss der Aufrufer der Write-Lock-Operation eine Menge von Adressbereichen angeben. Aus dieser Menge von Adressbereichen wird die Menge der zeitlich nächstliegenden Write-Lock-Instanzen bestimmt, von denen eine direkte kausale Abhängigkeit angenommen werden soll. Diese Art der Abhängigkeit dient lediglich als Informationsübergabe für zusätzliche Bedingungen oder Eigenschaften wie beispielsweise die *Serialisierbarkeit*, deren Einhaltung eine Frage konkreter Implementierungs-Strategien ist. Die folgende Situation lässt sich beispielsweise auf verschiedene Arten lösen:

## 7.6 Aktualitätsgrade

Der aufmerksame Leser wird sich bereits gefragt haben, wie *unlock*-Operationen in einem Modell funktionieren sollen, das alle Lock-Instanzen als (quasi-)persistente Objekte auffasst und sie im Extremfall für immer aufbewahren kann (womit z.B. in militärischen oder geheimdienstlichen Anwendungen jederzeit nachvollziehbar wäre, wer wann welche Read-Locks gesetzt und die zugehörigen Daten inspiert hat).

Die als Container für Zugriffs-Operationen wirkenden Lock-Instanzen müssen irgendwann *abgeschlossen* werden. Abschließen bedeutet, dass der Container keine (nachträglich angelieferten) Operationen mehr für die Beherbergung akzeptiert.

Beim Anlegen wird jedem Container ein Mandat oder eine Menge von Mandaten als „Eigentümer“ des Containers

<sup>9</sup>Dies ist ähnlich zu nicht-determinanten Zustands-Abhängigkeiten aus Abschnitt 3.2, bezieht sich jedoch nicht auf die Zustände selbst, sondern auf die in den Zuständen gespeicherten kausalen Abhängigkeiten.

- zugeordnet. Der *Zustand* eines Containers wird beim Anlegen ebenfalls festgelegt, kann jedoch später geändert werden und ist ein Wert aus der folgenden hierarchisch geordneten Aufzählung von Aktualitätsgraden:

**act\_invalid** Der Container beherbergt ungültige (d.h. zu ignorierende) Operationen.

**act\_destroyable** Der Container beherbergt ein vorläufiges Berechnungsergebnis, das jederzeit (evtl. asynchron) auch durch fremde Mandate (nicht nur durch das Eigentümer-Mandat) für ungültig erklärt werden kann, ohne dass dadurch ein irreparabler Schaden entsteht.

**act\_optimistic** Der Container beherbergt eine vorläufige Version, deren Gültigkeit noch nicht feststeht. Eine Ungültigkeits-Erklärung kann jedoch nur durch das Eigentümer-Mandat erfolgen (vgl. optimistische Strategien in der Datenbank-Literatur [VGH93]).

**act\_temp** Es handelt sich um einen vorläufigen (noch nicht abgeschlossenen) Container, zu dem jederzeit neue Operationen zur Beherbergung hinzukommen können. Ein Entzug bereits beherberter Operationen ist im Normalfall nicht vorgesehen (einige Ausnahme: Systemfehler oder andere katastrophale Ereignisse).

**act\_close** Der Container ist geschlossen, d.h. es können keine neuen Operationen mehr beherbergt werden. Die Operationen bzw. deren Effekte brauchen nicht unbedingt persistent gespeichert zu sein. Ein erneutes Öffnen, d.h. der Wechsel in einen kleineren Zustand mittels *lock* ist dann (und nur dann) zulässig, wenn nirgendwo kausale Abhängigkeiten von diesem Container bestehen.

**act\_freeze** Wie *act\_close*, jedoch ist ein Wechsel in kleinere Zustände nicht mehr möglich.

**act\_multisafe** Wie vorher, der Container samt beherbergtem Container-Inhalt ist jedoch mindestens 1 Mal persistent gespeichert und daher auch nach einfachen Systemfehlern wie Stromausfall noch erreichbar.

**act\_multisafe** Der Container(-Inhalt) ist mehrfach redundant gegen schwerere Systemfehler gesichert.

Für die Zustände gelten folgende Regeln: Anlegen eines Containers ist nur mit den Aktualitätsgraden *act\_destroyable* bis *act\_tmp* möglich. Eine *unlock*-Operation setzt den Zustand aller von ihrem Adressbereich überstrichenen und im Mandats-Eigentum befindlichen Container auf einen wählbaren anderen Aktualitätsgrad; Zustands-Änderungen geschehen durch *unlock* mit Ausnahme von *act\_invalid* nur in aufsteigender Richtung der Aktualitätsgrade. Der Zustand *act\_invalid* ist von *act\_destroyable*

bis *act\_close* aus erreichbar (also insbesondere ab *act\_freeze* nicht mehr erreichbar); ein einmal erreichter *act\_invalid*-Zustand kann nie mehr verlassen werden.

Weiterhin sollte beachtet werden, dass Zustandswechsel in höhere Aktualitätsgrade nur dann sinnvoll sind, wenn alle kausal vorangehenden Container mindestens den gleichen Aktualitätsgrad erreicht haben. Falls ein kausal vorangehender Container auf *act\_invalid* gesetzt wird, müssen alle davon kausal abhängigen Container ebenfalls auf *act\_invalid* gesetzt werden. Dies entspricht den *kaskadierenden Rollbacks* in klassischen Transaktions-Modellen.

Die *unlock*-Operation kann damit sowohl die klassische Funktionalität von Rollback-Operationen von Transaktionen als auch von Commit-Operationen ausführen. Ein Rollback entspricht dem Zustand *act\_invalid*, beim Commit wird je nach geforderter Fehlertoleranz ein Aktualitätsgrad ab *act\_freeze* angesteuert.

## 7.7 Schnittstellen von Lock-Operationen

### 7.7.1 Suchintervalle

Ein Nest lässt sich im multiversion-Modell als zweidimensionalen Raum auffassen, der eine örtliche und eine zeitliche Dimension hat. Prinzipiell kann man jederzeit auf alle Teile dieses Raumes zugreifen. Dazu gibt es Suchoperationen, mit denen sich Lock-Container im Nest aufspüren lassen. Zur Spezifikation von Suchintervallen schlage ich folgende Konventionen vor:

Ein *Suchintervall* ist ein Paar  $(start, delta)$ , wobei *start* den Beginn und *start + delta* das Ende des Such-Bereiches angeben. Ein Suchintervall kann sich entweder auf die zeitliche oder auf die örtliche Dimension beziehen. Bei positivem *delta* wird in aufsteigender Richtung gesucht, d.h. falls sich mehrere Lock-Container mit dem Suchintervall in der jeweiligen Dimension schneiden, wird nach dem Durchführen von Vergleichsoperationen und eventuell davon ausgelöster Verkleinerung (nur bei der zeitlichen Dimension) diejenige Lock-Instanz ausgeliefert, die den kleinsten Wert in der jeweiligen Dimension hat und sich mit dem Suchintervall überschneidet. Bei negativem *delta* wird in umgekehrter Richtung gesucht.

Bie gleichzeitiger Suche nach Suchintervallen *adr\_search* und *time\_search* werden diejenigen Lock-Instanzen ausgeliefert, die sich in beiden Dimensionen mit den jeweiligen Suchintervallen überschneiden. Auf diese Weise ist eine Suche in einem rechteckigen Bereich des Ort-Zeit-Raumes möglich.

Als Sonderfall kann weiterhin  $delta = +\infty$  oder  $delta = -\infty$  zur unbeschränkten Suche zugelassen werden, ebenso  $start = +\infty$ .

### 7.7.2 Lock-Operationen im Multiversion-Modell

Die *lock*-Operation aus Abschnitt 3.3.5 wird im hier vorgestellten Erweiterungs-Vorschlag auf multiversion-Modelle aus methodischen Gründen in die bei-

den Operations-Varianten `read_lock` und `write_lock` aufgesplittet, da sie eine leicht unterschiedliche Parameter-Versorgung benötigen. Es sind jedoch auch geringfügig andere Gestaltungen der Schnittstelle möglich, die die Uniformität der Schnittstelle beider Lock-Arten beibehalten.

```
read_lock(nest, mandate,
          speculative_set, adr_set, time_set,
          act, min_act, stable, direct, aim, action) →
          (adr_result, time_result)
```

Die Parameter `speculative_set` und `adr_set` übernehmen die Funktion von `log_address`, `len`, `try_address` und `try_len` aus Abschnitt 3.3.5. Zusätzlich wird ein Vorschlag für das Zeitstempel-Intervall `time_set` übergeben. In den Ergebnis-Intervallen `adr_result` und `time_result` werden die tatsächlich möglichen Orts- und Zeit-Intervalle zurückgeliefert (sofern das Setzen des Locks überhaupt möglich ist). Das Orts-Intervall muss auf jeden Fall mindestens `adr_set` umfassen, das Zeit-Intervall kann gegenüber `time_set` auch verkleinert ausfallen.

Der Parameter `act` gibt den Aktualitätsgrad des zu erzeugenden Read-Locks vor. `min_act` gibt dagegen einen Aktualitätsgrad vor, den sämtliche direkt kausal vorangehenden Write-Locks *mindestens* erfüllen müssen. Falls sich in der direkten kausalen Abhängigkeit Write-Locks mit zu geringem Aktualitätsgrad befinden, muss im Regelfall so lange gewartet werden, bis diese den geforderten Aktualitätsgrad erreicht haben. Wenn man kleine Aktualitätsgrade für `min_act` vorgibt, wird zwar die mögliche Parallelität gesteigert (vgl. „dirty read“ in Datenbanken), gleichzeitig steigt jedoch das Risiko kaskadierender Rollbacks und auch das Risiko von inkonsistentem Lesen durch neu hinzukommende Beherbergungen in kausal vorgehenden Containern. Bei strengen Anforderungen an einzuhaltende Konsistenz-Bedingungen kann damit u.U. die Notwendigkeit von Rollbacks impliziert werden (vgl. *Fehlersicherheit von Schedules* [VGH93]).

Die kausalen Abhängigkeiten werden im allgemeinsten multiversion-Modell persistent aufbewahrt, um die Speicher-Semantik auch bei später nachgelieferten „vordatierten“ Locks erfüllen zu können. Der boolsche Parameter `stable` gibt vor, ob die kausalen Abhängigkeiten „stabil“ sein sollen, oder ob sich spätere Locks „dazwischen schieben“ dürfen. Bei `stable = true` dürfen später im Zeit-Intervall zwischen dem neu gesetzten Lock und dem kausal abhängigen Vorgänger keine solchen Write-Locks mehr nachträglich eingebaut werden, die zu *irgendeiner* Veränderung des Suchergebnisses bei der (erneuteten) Bestimmung der funktionalen Abhängigkeiten aller möglicherweise betroffenen Lock-Instanzen führen können<sup>10</sup>. Dadurch bestimmt der *Eigentümer* eines Locks (d.h. das Mandat oder eine Menge von Mandaten), ob andere Locks mit ihm verträglich sind oder nicht. Wenn man diese Eigenschaft auch transitiv für alle Vorgänger der kausalen Ab-

<sup>10</sup>Die Bestimmung aller möglicherweise betroffenen Lock-Instanzen muss nicht bei jedem Setz-Versuch neu erfolgen, wenn das Prinzip des *dynamic programming* angewandt wird. Weiterhin darf eine Implementierung das spätere Zwischenschieben von Locks auch dann zurückweisen, wenn dies laut `stable`-Angaben möglich wäre. Dadurch lassen sich Heuristiken einführen, die z.B. relativ grosse Bereiche der Raum-Zeit-Ebene gegen das Setzen neuer Write-Locks sperren.

hängigkeiten ermöglicht (z.B. durch einen Aufzählungstyp mit `stable = transitive`; die Bestimmung gilt unabhängig vom ursprünglichen `stable`-Zustand der transitiv vorangehenden kausalen Abhängigkeiten), dann lassen sich verschiedene Transaktions- oder Konsistenz-Modelle beliebig miteinander mischen, ohne die jeweils intendierte Semantik zu verletzen (dies steht im Gegensatz zur herrschenden Meinung über die Mischbarkeit verschiedener Modelle).

Der boolsche Parameter `direct` bestimmt, ob die erstellte kausale Abhängigkeit unbedingt vom *ersten* tatsächlich vorhandenen Write-Lock in Suchrichtung abhängen muss, oder ob (z.B. in verteilten Systemen) veraltete Versionen geduldet werden. Letztere Möglichkeit verbessert die Performanz zu Lasten weniger strikter Konsistenz-Modelle und kann zu „überspringenden“ kausalen Abhängigkeiten ähnlich der PRAM-Konsistenz<sup>11</sup> führen.

Der Parameter `aim` dient als Ersatz für den früheren Parameter `kind` aus Abschnitt 3.3.5. Er gibt einen der Werte `aim_address`, `aim_data` oder `aim_both` an. Damit wird spezifiziert, ob nur die Adress-Abbildungs-Funktion des Nestes, oder die Daten-Abbildungs-Funktion, oder beide gegenüber konkurrierenden Zugriffen gesperrt werden sollen (weiterhin kann damit auch die Zulässigkeit der beherbergten Operations-Arten geprüft werden). Adressmodifizierende Operationen wie `move` lassen sich dadurch von datenmodifizierenden Operationen wie `transfer` entkoppeln. Um diese Fähigkeit besser auszunutzen, kann man beispielsweise einen weiteren Parameter `adr_time` einführen, der den zu verwendenden Zeitstempel für die Adressabbildung unabhängig von der Datenabbildung vorgibt. Damit können insbesondere solche Mandate einen ungestörten Zugriff auf die Nest-Adressen durchführen, die selbst keine adressmodifizierenden Operationen ausführen; diesen Mandaten erscheint dann die Adress-Abbildung als „festgenagelt“, obwohl sie von anderen Mandaten auf der virtuellen Zeitachse geändert worden sein kann.

Für die Ortsangabe `adr_set` gilt ähnlich zu Abschnitt 3.3.5 die Regel, dass für das gleiche Mandat früher gesetzte Locks (egal wo sich diese auf der Zeit-Achse befinden) „verdeckt“ werden können. Jedes Mandat hat zu jedem Zeitpunkt höchstens eine (eindeutige) Lock-Instanz-Zuordnung für jeden (Teil-)Orts-Bereich; diese eindeutige Zuordnung dient weiterhin der korrekten Versions-Zuordnung nachfolgender `get`- und `transfer`-Operationen. Die Eindeutigkeit wird dadurch erzielt, dass bei nochmals angeforderten Adressbereichen die alten Versionen bei Bedarf (d.h. nur bei geändertem Zeit-Bereich) ganz oder teilweise *aufgegeben* werden, d.h. aus der Eigentümer Zuordnung entfernt werden (wodurch sie quasi „Allgemeingut“ werden). Aufgegebene Versionen, die nicht abgeschlossen sind, werden standardmäßig so behandelt, als würden sie atomar durch `unlock` mit Aktualitätsgrad `act_invalid` (bei nicht-transaktionalen Untermodellen evtl. auch mit `act_freeze`) freigegeben und sofort wie-

<sup>11</sup>Weitere Konsistenzmodelle, insbesondere kausal halbgeordnete (z.B. [BSS91]), lassen sich ggf. im `direct`-Parameter durch Ersatz des Typs `boolean` durch einen Aufzählungstyp spezifizieren, der Zwischenstufen zwischen Ordnungslosigkeit und Totalordnung kausaler Abhängigkeiten ausdrücken soll.

- der an der neuen Zeit-Stelle neu gesetzt. Auf diese Weise ist sicher gestellt, dass Adressangaben bei get oder anderen Operationen stets eine eindeutig bestimmte Version eines Locks bezeichnen und das „Vererben“ des Zeitstempels vom Container auf die beherbergten Operations-Instanzen eindeutig ist.

```
write_lock(nest,mandate,
           speculative_set,adr_set,time_set,
           act,causal,aim,action) →
           (adr_result,time_result)
```

Gegenüber `read_lock` kommt hier die direkte Angabe einer Menge (bzw. Liste oder Array) von Quadrupeln `causal` hinzu. Ein Quadrupel hat die Form `(adr_search, min_act, stable, direct)`. Der Wert von `adr_search` spezifiziert den Ort, an dem die kausale vorgehenden Read- oder Write-Locks liegen. Durch die Aufnahme von `min_act, stable` und `direct` in die Quadrupel lassen sich diese Eigenschaften für die funktionalen Abhängigkeiten in jedem Adressbereich einzeln einstellen<sup>12</sup>.

Falls im Suchbereich bereits ein Lock-Container mit Aktualitätsgrad `act_close` existiert, dessen bereits vorhandene kausale Abhängigkeiten eine kompatible Obermenge zu den geforderten darstellen und der selbst nicht an späteren kausalen Abhängigkeiten teil nimmt, dann darf dieser Container an Stelle einer Neuerzeugung eines Containers erneut geöffnet werden. Diese Optimierung braucht nicht in jeder Implementierung unterstützt zu werden; `act_close` darf beim `unlock` auch als `act_freeze` behandelt werden.

```
unlock(nest,mandate,adr_search,act)
```

Gegenüber der früheren Version aus Abschnitt 3.3.5 kommt hier lediglich der Aktualitätsgrad `act` hinzu, auf den die freizugebenden Locks gesetzt werden sollen. Es ist möglich, Locks mehrmals schrittweise auf immer höhere Aktualitätsgrade zu setzen, so dass eine korrekte Klammerung von `lock / unlock`-Paaren nicht mehr eingehalten werden muss.

```
notify_lock(nest,mandate,adr_try,adr_need,
            time_need,aim,urgency) → success
```

Bei einer (asynchrone) Rückgabe eines nicht-abgeschlossenen Locks. Neben der Rückgabe spekulativ vergrößerter Adressbereiche lässt sich über `time_need` auch die Verkleinerung von Zeit-Intervallen in verteilten Systemen konsistent durchführen, wenn die Konvention herrscht, dass nicht abgeschlossene Locks nur von ihrem Eigentümer verkleinert werden dürfen.

<sup>12</sup>Falls diese individuelle Einstellungsmöglichkeit nicht benötigt wird, kann man sie auch in die Parameter von `lock` verschieben und dadurch die Schnittstelle vereinfachen. Welche Semantik sich durch individuelle Einstellungen ergeben soll und ob diese im Sinne eines bestimmten semantischen Modells überhaupt gültig ist oder zurückgewiesen werden muss, kann auch (teilweise) in der konkreten Baustein-Implementierung der Bearbeiter-Instanz festgelegt werden. Ansonsten hat der Aufrufer die Auswahl aus einer ungeheueren Vielzahl von Semantiken, die er durch seine Parameter-Wahl vorgeben kann. Als sehr einfaches Beispiel lässt sich durch eine leere Menge von Quadrupeln spezifizieren, dass keine funktionalen Abhängigkeiten bestehen; dies kann bei der Initialisierung oder beim Löschen / unbedingten Überschreiben des Dateninhaltes durchaus vorkommen und zusammen mit der Rückdatierungs-Möglichkeit auf der virtuellen Zeitachse die Parallelität dieses evtl. langdauernden Vorgangs mit anderen Aktivitäten steigern.

## 7.8 Simulation einiger Transaktions-Strategien

Verschiedene Varianten klassischer ACID-Transaktionen werden durch die soeben vorgestellten Lock-Operationen auf folgende Weise simuliert:

Jeder einzelnen Schreib- und Lese-Operation, die sich auf einen noch nicht gelockten Adressbereich bezieht, werden Locks vorangestellt. Read-Locks werden bei Bedarf dadurch in Write-Locks aufgewertet, dass auf dem gleichen Adressbereich die andere Lock-Art, jedoch mit dem gleichen Zeit-Intervall angefordert wird. Ansonsten werden einmal angeforderte Locks bis zum Transaktions-Ende nicht mehr frei gegeben (2-Phasen-Locking).

Die kausalen Abhängigkeiten werden nach der Grundidee der Herbrand-Semantik syntaktischer Transaktions-Modelle [VGH93] stets so gesetzt, dass sämtliche von einer Transaktion jemals angefassten Adressbereiche bei den Write-Locks angegeben werden müssen.

Beim Commit werden alle gesetzten Locks atomar durch `unlock` auf dem gesamten Adressraum (d.h. man wählt `adr_search = (0,∞)`) mit mindestens `act = act_safe` abgeschlossen (sofern die D-Eigenschaft von ACID ausdrücklich bei jedem *einzelnen* Commit gefordert wird); beim Rollback ist stattdessen `act = act_invalid` zu wählen.

Bei den folgenden Simulationen konventioneller Transaktionen wird stets `stable = true` und `direct = true` gewählt. Die einzelnen Transaktions-Varianten unterscheiden sich dadurch, welche Zeit-Intervalle und `min_act`-Werte jeweils beim Setzen der Locks gewählt werden.

### 7.8.1 Timestamp-Ordering-Protokolle

Bei der Simulation von Multiversion-Timestamp-Ordering (MVTO) aus der Datenbank-Literatur [VGH93] wird die Serialisierbarkeit dadurch sicher gestellt, dass alle Operationen einer Transaktion als mit dem gleichen Zeitstempel durchgeführt gelten; dadurch entsteht eine virtuell zeitliche Totalordnung aller Operationen, die genau der Serialisierungs-Reihenfolge entspricht. Datenbanken benötigen hierzu ebenfalls die Fähigkeit zur Verwaltung mehrerer Versionen von Objekten (manchmal auch Time-Domain-Addressing genannt [GR93]). Der Zeitstempel wird bei Beginn der Transaktion oder beim ersten Zugriff einmal so bestimmt, dass jede Transaktion einen eindeutigen Zeitstempel besitzt, der nie mehr geändert wird.

Die Simulation von Schreib-Lese-Transaktionen durch multiversion-Nester ist relativ einfach: die Locks werden mit `time_set = (start,0)` und `act = act_tmp` gesetzt, wobei `start` den einmal bei Beginn der Transaktion eindeutig vergebenen Zeitstempel bedeutet.

Nur-Lese-Transaktionen lassen sich dadurch simulieren, dass bei Beginn der Transaktion abweichend von der oben beschriebenen allgemeinen Strategie ein einziger Read-Lock auf dem gesamten Adressbereich `adr_set = (0,∞)` gesetzt wird, so dass die anschliessenden Lese-Operationen nicht mehr einzeln durch Read-Locks abgesichert werden müssen. Ein derartiger Read-Lock, der den

gesamten Adressbereich umfasst, erzeugt einen *Schnappschuss*<sup>13</sup> des gesamten Nestes.

Da auch bei Schreib-Lese-Transaktionen sämtliche Read-Locks mit dem gleichen Zeitstempel gesetzt werden, lassen sich auch hier die individuellen Read-Locks durch einen Schnappschuss-Lock ersetzen; beim Schreiben werden dann einzelne Teile des Schnappschuss-Locks durchlöchert und zu Write-Locks aufgewertet.

### 7.8.2 Striktes 2-Phasen-Locking

Bei der Simulation klassischer 2-Phasen-Locks [VGH93] wird das multiversion-Verhalten gar nicht ausgenutzt. Read- und Write-Locks werden auf dem Zeitintervall `time_set = (now, ∞)` gesetzt, wobei now der streng monoton fortschreitenden aktuellen realen Uhrzeit beim Aufruf entspricht. Bei Gewähren des Locks wird dieses Zeitintervall auf den Gewährungs-Zeitpunkt verkleinert, so dass nachfolgende Write-Locks auf der Zeitachse Platz finden. In den Quadrupeln der Write-Locks wird durch `min_act = act_freeze` sicher gestellt, dass zu jedem Zeitpunkt höchstens ein einziger nicht abgeschlossener Write-Lock auf überlappenden Adressen existiert.

### 7.8.3 Weitere Arten von Serialisierbarkeit bzw. Konsistenzmodellen

Es sind auch mehrere Write-Locks zum gleichen Real-Zeitpunkt möglich, die sich lediglich nicht auf der virtuellen Zeitachse überschneiden dürfen. Man könnte dies mit Multiversions-Locking bezeichnen. Die Möglichkeit, überlappende Adressbereiche zum gleichen Real-Zeitpunkt sperren zu können, geht jedoch über den Begriff „multiversion concurrency control“<sup>14</sup> [HR01] bzw. den Begriff der Multiversions-Konflikt-Serialisierbarkeit [VGH93] hinaus. Eventuell wäre Time-Domain-Locking eine gute Bezeichnung zumindest für die englischsprachige Literatur.

Der verallgemeinerte Serialisierungs-Begriff aus [Vid87], der sowohl die Konflikt- als auch die MVTO-Serialisierbarkeit als Spezialfall enthält, dürfte sich durch die hier vorgestellte Nest-Schnittstelle ebenfalls simulieren lassen. Dies gilt ebenso für Protokoll-Familien wie MVS-GT (Multiversions-Serialisationsgraphen-Tester), die sich als konkrete Strategien in Bausteinen realisieren lassen.

Geschachtelte Transaktionen lassen sich durch Bausteine realisieren, die Gruppen von Änderungs-Operationen nach aussen hin durch gesammelte Lock-Operationen weitergeben und dabei ggf. die Aktualitätsgrade in andere Werte transformieren.

Durch „überspringende“ kausale Abhängigkeiten, Aufweichen der Herbrand-Semantik und Variation der `min_act`-, `stable`- und `direct`-Parameter lassen sich weitere Serialisierbarkeits-Begriffe bilden, deren systematische Untersuchung weiteren Arbeiten überlassen

<sup>13</sup> Im Unterschied zu [CL85] lässt sich ein Schnappschuss auch *rickwinkend* in der virtuellen Zeitachse machen.

<sup>14</sup> Bisher wurde Versionierung in Datenbanken offenbar nur zur Entkopplung zwischen Lesern und Schreibern eingesetzt. Das hier vorgestellte Modell ermöglicht dagegen prinzipiell auch die Entkopplung zwischen parallel aktiven Schreibern; Einschränkungen ergeben sich lediglich aus den kausalen Abhängigkeiten.

bleibt. Ein grosses Forschungs-Feld stellen Optimierungen bei sich widersprechenden Zielen dar, beispielsweise wenn spekulativ vergrößerte Lock-Bereiche nur mit „überspringenden“ kausalen Abhängigkeiten oder geringerem Aktualitätsgrad erkauf werden können<sup>15</sup> u.v.m.

Durch gezieltes Freigeben einiger Locks vor Transaktions-Ende lassen sich beliebige Misch-Paradigmen zwischen vollkommener transaktionaler Isolation und wechselseitigem Ausschluss realisieren<sup>16</sup>. Der bisher in Betriebssystemen verwendete wechselseitige Ausschluss wird durch das Konzept der Aktualitätsgrade so orthogonal ergänzt, dass ein nachträgliches Rücksetzen schiefgeganger Berechnungen (die mit Berechnungen anderer Mandate nicht-serialisierbar verbunden sein können) möglich ist.

### 7.8.4 Automatische Re-Evaluation

Die Idee besteht darin, in den Containern (gelegentlich oder teilweise auch in den gespeicherten kausalen Abhängigkeiten) nicht (nur) die Ergebnisse der von den Konsumenten generierten Zugriffs-Operationen zu beherbergen, sondern (zusätzlich) ihre *Berechnungsvorschrift*.

Dadurch kann man die beherbergten Zugriffs-Operationen neu berechnen, wenn nachträgliche Änderungen bei einem der kausal vorgehenden Container oder nachträgliches Einschieben von Write-Locks in stabile kausale Abhängigkeits-Beziehungen zugelassen werden, wodurch die kausalen Abhängigkeiten auf die neuen Versionen abgeändert werden. Bei solchen Änderungen wird lediglich eine Neuberechnung des Container-*Inhaltes* aller transiv kausal abhängigen Container angestoßen; die Container selbst und ihre Position in der Raum-Zeit-Ebene bleiben erhalten.

Eine Spezifikation der Berechnungsvorschrift kann durch Einführen eines geeigneten *Interpreters* und einer *Interpreter-Sprache* erfolgen, von der Programmstücke mittels einer Operation `download` bekannt gemacht werden (z.B. zur Addition von Zahlen-Werten oder Berechnung einer Summe). Das Herunterladen von ausführbarem Maschinencode ist prinzipiell ebenfalls möglich, hat jedoch Nachteile bei der Sicherheit (zu deren Lösung kann man spezielle Konventionen für die während der Evaluation aktiven Schutzbereiche einführen).

In beiden Fällen sollten möglichst *alle* Konsumenten ihre *transfer*-Operationen zum Ändern der Daten- und/oder Adressabbildung von Nestern durch `download`-Operationen ersetzen, damit nicht-reevaluierbare Container dieses Verfahren nicht stoppen können oder inkorrekt werden lassen.

<sup>15</sup> Die klassischen Flaschenhals-Probleme verteilter Systeme, insbesondere die durch Latenzen verursachten, lassen sich ebenfalls unter dem Aspekt sich widersprechender Optimierungs-Ziele (Tradeoffs) betrachten. Interessant wäre eine Vereinheitlichung der Darstellung dieses Gebiets mit den hier aufgeworfenen Tradeoffs.

<sup>16</sup> Eine griffige Formel zur Charakterisierung beider Extreme könnte auf dem Hintergrund der Betrachtung von Locks als Container auf der virtuellen Zeitachse folgendermassen lauten: Bei transaktionaler Semantik werden Container so lange wie möglich offen gehalten (um sie ggf. für invalide erklären zu können), beim wechselseitigen Ausschluss werden Container dagegen so früh wie möglich geschlossen.

## Anhang A

# Simulation von Vererbung durch Generizität

Zum Nachweis, dass sich Erweiterungs-Generizität mit Hilfe von Parametrischer Generizität ausdrücken lässt, simulieren wir die Kernidee der objektorientierten Vererbung durch Präprozessor-Makros der Sprache C:

```
// Allgemeine Deklarationen
#define class struct

List iterators
#define decl_method(base,type,name) base##_##name##_head((*base##_##name),class type);
#define def_method(base,type,name) \
    static base##_##name##_head(base##_##name,class type) \
    base##_##name##_body(class type)
#define init_item(base,type,name) self->type##_##name = type##_##name;

// Hilfs-Makros
#define init_head(type) void type##_init(class type * self)
#define declarations(type) type##_attribs type##_list(decl_method,type,type)
#define inherit(base,type) base##_attribs base##_list(decl_method,base,type)
#define inherit_init(base,type) base##_init((class base*)self);

// Haupt-Makros
#define base_class(type) \
class type;
type##_list(def_method,type,type)
class type {
    declarations(type)
};

t_head(type)
type##_list(init_item,type,type)

#define sub_class(base,type) \
class type;
type##_list(def_method,type,type)
class type {
    inherit(base,type)
    declarations(type)
};

init_head(type)
{
    inherit_init(base,type)
    type##_list(init_item,base,type)
}
```

Auf diese allgemeinen Deklarationen hin folgt nun die einmalige Angabe zweier Klassen A und B mittels Makro-Definitionen:

```

// spezifisch fuer A

#define A_attributes
  int member1;
  int member2;

#define A_method1_head(name,type) \
void name(type * self)

#define A_method1_body(type) \
{
  /*...*/
}

#define A_method2_head(name,type) \
void name(type * self)

#define A_method2_body(type) \
{
  /*...*/
}

#define A_list(op,b,t)      op(b,t,method1) op(b,t,method2)

// Anwendung A

base_class(A)

// spezifisch fuer B

#define B_attributes
  int member3;
  int member4;

#define B_method3_head(name,type) \
void name(type * self)

#define B_method3_body(type) \
{
  /*...*/
}

#define B_list(op,b,t)      op(b,t,method3)

// Anwendung B

sub_class(A,B)

```

Mit Hilfe der Unix-Kommandos `gcc -E vererbung.c | grep -v '^#' | indent -bad -bap | indent -kr` entsteht hieraus der folgende expandierte Quelltext:

```

struct A;
static void A_method1(struct A *self)
{
}
static void A_method2(struct A *self)
{
}
struct A {
  int member1;
  int member2;
  void (*A_method1) (struct A * self);
  void (*A_method2) (struct A * self);
};
void A_init(struct A *self)
{
  self->A_method1 = A_method1;
  self->A_method2 = A_method2;
}
struct B;
static void B_method3(struct B *self)
{
}
struct B {

```

## ANHANG A. SIMULATION VON VERERBUNG DURCH GENERIZITÄT

```
int member1;
int member2;
void (*A_method1) (struct B * self);
void (*A_method2) (struct B * self);
int member3;
int member4;
void (*B_method3) (struct B * self);
};

void B_init(struct B *self)
{
    A_init((struct A *) self);
    self->B_method3 = B_method3;
}
```

## Anhang B

# Beispiel-Probleme der Objektorientierung

Ein Beispiel, bei dem die Mitvererbung von Datenstrukturen bzw. Repräsentationen zu einer kombinatorischen Explosion von Klassen führen kann:

Die Erweiterung einer abstrakten Basisklasse  $G$  solle auf zwei verschiedene voneinander unabhängige Arten stattfinden. Dies lässt sich relativ einfach durch zwei verschiedene abgeleitete abstrakte Basisklassen  $A$  und  $B$  durchführen. Aus jeder der beiden abstrakten Basisklassen seien verschiedene konkrete Klassen  $A_1, \dots, A_n$  und  $B_1, \dots, B_m$  abgeleitet, deren Funktionalität jeweils benötigt werde. Nun komme die Anforderung hinzu, dass die Vereinigung beider Schnittstellen ebenfalls benötigt werde. Auf der Ebene der abstrakten Basisklassen ist dies sehr einfach durch eine weitere Basisklasse  $C$  zu bewerkstelligen, die ihre Schnittstelle aus  $A$  und  $B$  durch Mehrfachvererbung erhält. Dies bedeutet für die davon abgeleiteten konkreten Klassen  $C_{ij}$  jedoch, dass nun das gesamte kartesische Produkt aus  $n \cdot m$  abgeleiteten Klassen in Frage kommt und ggf. zu implementieren ist. Selbst wenn dabei keine neue Funktionalität mittels überschriebener Methoden-Implementierungen hinzukommen sollte, führt dies zu einer kombinatorischen Explosion der Klassen-Namen und zu einer Aufblähung des gesamten Systems. Nach meinem Dafürhalten taucht dieses Problem oft in der Praxis auf, ohne erkannt zu werden.

Für dieses Problem gibt es eine oft funktionierende Lösung, ohne das objektorientierte Paradigma verlassen zu müssen. Die Implementierung einer abgeleiteten Klasse  $C_{ij}$  sollte nicht durch Mehrfachvererbung erfolgen, sondern durch Container-Instanzen  $c_{ij} \in C_{ij}$ , die jeweils zwei Instanzen  $a_i \in A_i$  und  $b_j \in B_j$  enthalten (in Form einer has-a-Beziehung statt einer is-a-Beziehung). Man kann  $C_{ij}$  auch als Proxy-Klasse bezeichnen. Dies bringt jedoch folgende Schwierigkeiten mit sich:

1. Die bisherigen Attribute waren Mitglieder der alten Objektinstanz-Repräsentation; von der neuen Repräsentation werden sie entweder indirekt adressiert oder sie werden nochmals eingebettet. Dadurch wird Platz verschwendet.
2. Die gemeinsame Oberklasse  $G$  kann Attribute enthalten, die bei der Mehrfachvererbung nur einmal gemeinsam instantiiert werden sollen.
3. Der Namensraum wird nach wie vor durch  $n \cdot m$  verschiedene Klassennamen verschmutzt.

Das erste Problem ist lästig, gefährdet aber die Korrektheit nicht. Zur Lösung des zweiten Problems kann man eine drit-

te Komponente  $g \in G$  in die Container-Instanz einführen. Das dritte Problem ist nicht zu umgehen und belastet im günstigsten Fall nur den Compiler.

Es gibt jedoch noch eine weitere Lösung: man führt eine Klasse  $C$  ein, die die Summe der Schnittstellen enthält und ansonsten nur als trivialer Container fungiert, der die eigentliche Implementierung in  $G$ ,  $A$  und  $B$  aufruft (was den Schnittstellen-Aufwand etwa verdoppelt). Damit betreibt man in der Objektorientierung jedoch letztlich die gleiche Art von Einbettung von Unterobjekt-Instanzen, wie sie in „konventionellen“ Programmierparadigmen schon immer gehandhabt wurde; es wird lediglich ein Zusatz-Aufwand mit Duplizierung von Schnittstellen getrieben, um das objektorientierte Paradigma mittels Mehrfachvererbung an der Schnittstelle retten zu können. Es stellt sich die Frage, ob die konzeptuell durch Mehrfachvererbung entstandene Schnittstelle  $C$ , deren Implementierung allerdings keine (Mehrfach-)Vererbung benutzt, wirklich benötigt wird, oder ob die durchgehende Benutzung von  $A$  und  $B$  als Ersatz für  $C$  nicht unter dem Strich einfacher ist. Unabhängig davon sehe ich das für die Objektorientierung wesentliche Paradigma der Vererbung nicht mehr als tatsächlich benutzt, und ich stelle die Frage, ob man es nicht gleich von vornherein hätte weglassen können.

Dies bedeutet nach meiner Ansicht nicht, dass die automatisierte Ableitung von objektorientierten Datenschemata aus Klassen-Schemata generell nutzlos ist, denn es gibt sehr wohl Anwendungsfelder, wo die automatische Ableitung immer wiederkehrende Arbeitsvorgänge rationalisiert.

Ich bin der Ansicht, dass die Objektorientierung ein oft nützliches Denkschema zum Entwurf von Schnittstellen ist; dass aber eine unreflektierte Umsetzung dieses Schemas in konkrete Datenstrukturen kostspielig oder sogar gefährlich werden kann, da die üblicherweise angebotenen Daten-Repräsentationen nicht für die Aufgabenstellung optimal zu sein brauchen.

Auf methodischer Ebene scheint mir das geschilderte Problem auch darin zu liegen, dass objektorientierte Entwurfs-Denkweise oftmals auf Biegen und Brechen versucht, alles mittels Erweiterungs-Generizität zu lösen. Die hier vorgeführte Problemstellung passt jedoch zur kompositorischen Generizität viel besser.

Dieses Beispiel ist daher ein weiteres Argument für den von mir propagierten Vorrang kompositorischer Generizität vor Erweiterungs-Generizität.

## Anhang C

# Ansätze zur Vermeidung von Wettrennen

1. Man sorgt dafür, daß stets genügend Ressourcen vorhanden sind und ein Entzug daher niemals notwendig wird. Leider spricht die Realität oft dagegen, da Ressourcen stets endlich sind und nach einem Quasi-Naturgesetz oft nicht zur Deckung des Bedarfs ausreichen. Wenn der Bedarf limitiert ist oder wenn man bereits bei der Ausgabe von Ressourcen Limits einführt (z.B. Quota), kann man u.U. auf Rückfordernsmechanismen verzichten. Ebenfalls in diese Kategorie fällt z.B. der Tausch von gleichwertigen Ressourcen, bei dem ebenfalls eine Rückforderung in Zukunft ausgeschlossen werden kann. Im Sinne des Eigentum / Besitz-Modells (Abschnitt 5.1) bedeutet dies, dass echtes Eigentum (nicht nur bloßer Besitz) übertragen wird, das nie mehr zurückgefordert werden kann.
2. Man verlangt, daß Besitz grundsätzlich nur an hierarchisch tieferstehende Instanzen und nur für die Dauer der Bearbeitung einer Operation weitergegeben werden darf; damit würde das Konzept des logischen IO nicht mehr benutzt und das Modell eingeschränkt.
3. Man erlaubt nur die Rückvergabe von solchem Unterbesitz an höhere Instanzen, der bereits einmal von der gleichen höheren Instanz als Hauptbesitz an die niedrigere vergeben wurde (transitive Rücklieferung von Unterobjekten). Damit gehört der Unterbesitz in Wirklichkeit stets der höheren Instanz; eine Rückforderung nach den Hauptbesitzer wird als *unberechtigt* klassifiziert und nicht ausgeführt bzw. gar nicht erst versucht. Diese Konstruktion erweitert das Modell gegenüber 2. ohne gravierende Nachteile einzuführen; daher gebe ich ihm den Vorzug gegenüber 2. Allerdings sehe ich es immer noch als relativ stark eingeschränkt an, insbesondere für Anwendungen in verteilten Systemen und Netzwerk-Betriebssystemen. Auf der konzeptuellen Ebene scheint mir eine Vertauschung von Etiketten statt zu finden, da in Wirklichkeit keine echte Delegation von Macht über den Besitz statt findet.
4. Man vergibt Besitz an höhere Instanzen (ggf. auch an niedrigere Instanzen) grundsätzlich nur auf begrenzte (Real-)Zeit, innerhalb deren die Ressource entweder zurückgegeben oder die Verleihzeit verlängert werden muß; die Zeitschranke wird so bemessen, daß ein Verfall des Leihdatums im Normalbetrieb nicht stattfindet. Nach diesem Modell ist es die Pflicht eines jedes Ausleihs, entweder für rechtzeitige Rückgabe oder für Fristverlängerung zu sorgen (analog zum Ausleih-Verfahren einer Universitäts-Bibliothek). Wenn die Zeitschranke überschritten wird, dann liegt in diesem Sinne ein *Fehlverhalten* vor, und die Ressourcen können z.B. automatisch entzogen werden. Dieser Entzug hat jedoch wieder den Charakter eines asynchron auftretenden Signals, d.h. es ändert nichts am Prinzip der Rückgabe, nur am Auslöser. Dieses Modell eignet sich insbesondere für *einige* Anwendungen in verteilten Systemen, da eine nicht rechtzeitig erfolgte Rückgabe bzw. Verlängerung auch als *Ausfall* einer Instanz interpretiert werden kann, auf den dann z.B. mit automatischen Entzug des Ressourcen-Besitzes und Neuverteilung *reagiert* wird; falls die Instanz nicht tatsächlich ausgefallen war, sondern nur die Kommunikationsverbindung, dann „weiß“ die betroffene Instanz nach Ablauf der Frist, dass sie die Verlängerung nicht mehr rechtzeitig geschafft hat und dass sie daher nicht mehr „rechtmäßiger“ Besitzer ist.
5. Die Besitzvergabe an höhere Instanzen geschieht ausschließlich unter der vollen Kontrolle der niedrigeren Instanz, die vollständig bestimmen kann, was damit geschieht. Damit wird jedoch das Hierarchie-Modell auch im regulären Betrieb auf den Kopf gestellt; es würde sich die Frage stellen, ob es noch eine Berechtigung hat. Auch die Eignung für verteilte Systeme scheint mir höchst zweifelhaft.

Keiner dieser Ansätze scheint mir geeignet, das Wettrennen-Problem (Kapitel 5) so aus der Welt zu schaffen, daß dadurch keine neuen Probleme entstehen und dass Anforderungen von verteilten Systemen ausreichend berücksichtigt werden. Eine Bewertung, welche dieser Probleme als gewichtiger zu betrachten ist, kann je nach Standpunkt des Bewerters und nach Anforderungen an das System unterschiedlich ausfallen (beispielsweise sehe ich bei extrem sicherheitskritischen Anwendungen Vorteile bei Modell 3). Es ist grundsätzlich möglich, eine konkrete Baustein-Hierarchie innerhalb der hier vorgestellten Architektur so zu gestalten, daß *notify\_\**-Operationen und deren Wettrennen vermieden werden; dies sehe ich dort als erstrebenswert an, wo eine Vermeidung ohne weiteres möglich ist und nur geringe Kosten verursacht. Ich sehe jedoch geringe Chancen, Wettrennen in einem verteilten System so zu vermeiden, daß keine Performanz-Nachteile oder Abhängigkeiten von der Funktionsfähigkeit zentraler Instanzen entstehen.

# Literaturverzeichnis

[AB86] ARCHIBALD, JAMES und JEAN-LOUP BAER: [BS96] *Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model.* Transactions on Computer Systems, 4(4):273–298, 1986.

[Ant90] ANTONOV, VADIM G.: *A Regular Architecture for Operating Systems.* Operating System Reviews, 24(3):22–39, 1990.

[Ass96] ASSENMACHER, HOLGER: *Ein Architekturkonzept zum Entwurf flexibler Betriebssysteme.* Dissertation Universität Kaiserslautern, 1996.

[AW94] ATTIYA, HAGIT und JENNIFER L. WELCH: *Sequential Consistency versus Linearizability.* Transactions on Computer Systems, 12(2):91–122, 1994.

[B+90] BERSHAD, BRIAN N. und OTHERS: *Lightweight Remote Procedure Call.* Transactions on Computer Systems, 8(1):37–5, 1990.

[B+95] BERSHAD, BRIAN N. und OTHERS: *SPIN – An Extensible Microkernel for Application-specific Operating System Services.* Operating System Reviews, 29(1):74–77, 1995.

[Bac86] BACH, MAURICE J.: *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[BHG87] BERNSTEIN, PILIP A., VASSOS HADZILACOS und NATHAN GOODMAN: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BN84] BIRRELL, ANDREW D. und BRUCE JAY NELSON: *Implementing Remote Procedure Calls.* Transactions on Computer Systems, 2(1):39–56, 1984.

[BPS81] BELADY, L. A., R. P. PARMELEE und C. A. SCALZI: *The IBM History of Memory Management Technology.* IBM Journal of Research and Development, 25(5):491–503, 1981.

[BR76] BLEVINS, PARKER R. und C. V. RAMAMOORTHY: *Aspects of a Dynamically Adaptive Operating System.* Transactions on Computers, 25(7):713–725, 1976.

[BS75] BERNSTEIN, ARTHUR J. und PAUL SIEGEL: *A Computer Architecture for Level Structured Systems.* Transactions on Computers, 24(8):785–893, 1975.

[BS96] BRUSTOLONI, JOSE CARLOS und PETER STEENKISTE: *Effects of Buffering Semantics on I/O Performance.* OSDI, Seiten 277–291, 1996.

[BS02] BROY, MANFRED und JOHANNES SIEDERSLEBEN: *Objektorientierte Programmierung und Softwareentwicklung.* Informatik Spektrum, 25(1):3–11, 2002.

[BSS91] BIRMAN, KENNETH, ANDRE SCHIPER und PAT STEPHENSON: *Lightweight Causal and Atomic Group Multicast.* Transactions on Computer Systems, 9(3):272–314, 1991.

[C+94] CHASE, JEFFREY S. und OTHERS: *Sharing and Protection in a Single-Address-Space Operating System.* Transactions on Computer Systems, 12(4):271–307, 1994.

[Che87] CHERITON, DAVID R.: *UIO: A Uniform I/O System Interface for Distributed Systems.* Transactions on Computer Systems, 5(1):12–46, 1987.

[CJ75] COHEN, ELLIS und DAVID JEFFERSON: *Protection in the Hydra Operating System.* Symposium on Operating System Principles, Seiten 141–160, 1975.

[CL85] CHANDY, K. MANI und LESLIE LAMPORT: *Distributed Snapshots: Determining Global States of Distributed Systems.* Transactions on Computer Systems, 3(1):63–75, 1985.

[Cla85] CLARK, DAVID D.: *The Structuring of Systems Using Upcalls.* Symposium on Operating System Principles, Seiten 171–180, 1985.

[CLG+93] CHEN, PETER M., EDWARD K. LEE, GARTH GIBSON, RANDY H. KATZ und DAVID A. PATTERSON: *RAID: High-Performance, Reliable Secondary Storage.* Computing Surveys, 26(2):145–185, 1993.

[CW85] CARDELLI, LUCA und PETER WEGNER: *On Understanding Types, Data Abstraction, and Polymorphism.* Computing Surveys, 17(4):471–522, 1985.

[Dat95] DATE, C. J.: *An Introduction to Database Systems.* Addison Wesley, 1995.

[Den68] DENNING, PETER J.: *The Working Set Model for Program Behavior.* CACM, 11(5):323–333, 1968.

- [Den71] DENNING, PETER J.: *Virtual Memory*. Computing Surveys, 2(3):153–189, 1971.
- [DGMS85] DAVIDSON, SUSAN B., HECTOR GARCIA-MOLINA und DALE SKEEN: *Consistency in Partitioned Networks*. Computing Surveys, 17(3):341–370, 1985.
- [DH66] DENNIS, JACK B. und EARL C. VAN HORN: *Programming Semantics for Multiprogrammed Computations*. CACM, 9(3):143–155, 1966.
- [Dij68] DIJKSTRA, EDGAR W.: *The Structure of the "THE" Multiprogramming System*. CACM, 11(5):341–346, 1968.
- [Dij71] DIJKSTRA, E. W.: *Hierarchical Ordering of Sequential Processes*. Acta Informatica, 1:115–138, 1971.
- [dJ93] JONGE, WIEBREN DE: *The Logical Disk: A New Approach to Improving File Systems*. Symposium on Operating System Principles, Seiten 15–28, 1993.
- [Doe96] DOEPPNER, THOMAS W.: *Distributed File Systems and Distributed Memory*. Computing Surveys, 28(1):229–231, 1996.
- [DS72] DENNING, PETER J. und STUART C. SCHWARTZ: *Properties of the Working-Set Model*. CACM, 15(3):191–198, 1972.
- [EKO95] ENGLER, DAWSON R., M. FRANS KAAS-HOEK und JAMES O'TOOLE: *Exokernel: An Operating System Architecture for Application-Level Resource Management*. Symposium on Operating System Principles, Seiten 251–266, 1995.
- [Elmbe] ELMAGARMID, AHMED K.: *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, ohne Jahresangabe.
- [Elmas] ELMASRI, RAMEZ und SHAMKANT B. NAVATHE: *Fundamentals of Database Systems*. Addison Wesley, 1995.
- [Esk96] ESKICIÖGLU, M. RASIT: *A Comprehensive Bibliography of Distributed Shared Memory*. Operating System Reviews, 30(1):71–96, 1996.
- [Fie88] FIELD, ANTHONY J.: *Functional Programming*. Addison-Wesley, 1988.
- [Fot61] FOTHERINGHAM, JOHN: *Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of Backing Store*. CACM, 4(10):435–436, 1961.
- [GC94] GOODHEART, BERNY und JAMES COX: *The Magic Garden Explained – The Internals of UNIX System V Release 4*. Prentice Hall, 1994.
- [GR93] GRAY, JIM und ANDREAS REUTER: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [Hab69] HABERMANN, A. N.: *Prevention of System Deadlocks*. CACM, 12(7):373–385, 1969.
- [Haj00] HAJJI, FARID: *Perl – Einführung, Anwendungen, Referenz*. Addison Wesley, 2000.
- [Han70] HANSEN, PER BRINCH: *The Nucleus of a Multiprogramming System*. CACM, 13(4):238–250, 1970.
- [Han73] HANSEN, PER BRINCH: *Concurrent Programming Concepts*. Computing Surveys, 5(4):223–245, 1973.
- [Hoa74] HOARE, C. A. R.: *Monitors: An Operating System Structuring Concept*. CACM, 17(10):549–557, 1974.
- [Hoa78] HOARE, C. A. R.: *Communicating Sequential Processes*. CACM, 21(8):666–677, 1978.
- [HP94] HEIDEMANN, JOHN S. und GERALD J. POPEK: *File-System Development with Stackable Layers*. Transactions on Computer Systems, 12(1):58–89, 1994.
- [HP95] HEIDEMÄNN, JOHN und GERALD POPEK: *Performance of Cache Coherence in Stackable Filing*. Symposium on Operating System Principles, Seiten 127–142, 1995.
- [HR83] HÄRDER, THEO und ANDREAS REUTER: *Principles of Transaction-Oriented Database Recovery*. Computing Surveys, 15(4):287–317, 1983.
- [HR01] HÄRDER, THEO und ERHARD RAHM: *Datenbanksysteme – Konzepte und Techniken der Implementierung*. Springer-Verlag, 2. Auflage Auflage, 2001.
- [Hur] THE GNU Hurd – GNU Project – Free Software Foundation (FSF). <http://www.gnu.org/software/hurd/hurd.html>.
- [HW90] HERLIHY, MAURICE P. und JEANNETTE M. WING: *Linearizability: A Correctness Condition for Concurrent Objects*. Transactions on Programming Languages and Systems, 12(3):463–492, 1990.
- [Jef85] JEFFERSON, DAVID R.: *Virtual Time*. Transactions on Programming Languages and Systems, 7(3):404–425, 1985.
- [JH02] JÄHNICHEN, STEFAN und STEPHAN HERRMANN: *Was, bitte, bedeutet Objektorientierung?* Informatik Spektrum, 25(4):266–275, 2002.

[Jon80] JONES, ANITA K.: *Capability Architecture Revisited*. *Operating System Reviews*, 14(3):33–35, 1980.

[Jür73] JÜRGENS, JÜRN: *Synchronisation paralleler Prozesse anhand von Zuständen*. Doktorarbeit, Technische Universität München, 1973.

[K<sup>+</sup>81] KAHN, KEVIN C. und OTHERS: *iMAX: A Multiprocessor Operating System for an Object-Based Computer*. Symposium on Operating System Principles, Seiten 127–136, 1981.

[K<sup>+</sup>91] KARLIN, A. und OTHERS: *Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor*. Symposium on Operating System Principles, Seiten 41–55, 1991.

[K<sup>+</sup>97] KAASHOEK, M. FRANS und OTHERS: *Application Performance and Flexibility on Exokernel Systems*. Symposium on Operating System Principles, Seiten 52–65, 1997.

[KE95] KLEIMAN, STEVE und JOE EYKHOLT: *Interrupts as Threads*. *Operating System Reviews*, 29(2):21–26, 1995.

[Knu94] KNUTH, DONALD E.: *The TeXbook*. Addison Wesley, 1994.

[LA93] LIM, BENG-HONG und ANANT AGARWAL: *Waiting Algorithms for Synchronization in Large-Scale Multiprocessors*. *Transactions on Computer Systems*, 11(8):253–294, 1993.

[Lag78] LAGALLY, K.: *Synchronization in a Layered System*. In: FLYNN, M. J. und OTHERS (Herausgeber): *Operating Systems, An Advanced Course*, Band 60 der Reihe *Lecture Notes in Computer Science*, Seiten 253–281. Springer-Verlag, 1978.

[Lam74] LAMPORT, LESLIE: *A New Solution of Dijkstra's Concurrent Programming Problem*. *CACM*, 17(8):453–455, 1974.

[Lam78] LAMPORT, LESLIE: *Time, Clocks, and the Ordering of Events in a Distributed System*. *CACM*, 21(7):558–565, 1978.

[Lam84] LAMPORT, LESLIE: *Using Time Instead of Timeout for Fault-Tolerant Distributed Systems*. *Transactions on Programming Languages and Systems*, 6(2):254–280, 1984.

[Lan81] LANDWEHR, CARL E.: *Formal Models for Computer Security*. *Computing Surveys*, 13(3):247–278, 1981.

[LCC<sup>+</sup>75] LEVIN, R., E. COHEN, W. CORWIN, F. POLLACK und W. WULF: *Policy/Mechanism Separation in Hydra*. Symposium on Operating System Principles, Seiten 132–140, 1975.

[LH89] LI, KAI und PAUL HUDA: *Memory Coherence in Shared Virtual Memory Systems*. *Transactions on Computer Systems*, 7(4):321–359, 1989.

[Lie95a] LIEDTKE, JOCHEN: *Address Space Sparsity and Fine Granularity*. *Operating System Reviews*, 29(1):87–90, 1995.

[Lie95b] LIEDTKE, JOCHEN: *On  $\mu$ -Kernel-Construction*. Symposium on Operating System Principles, Seiten 237–250, 1995.

[Lio96] LIONS, JOHN: *Lions' Commentary on UNIX 6th Edition with Source Code*. Peer-to-Peer Communications, 1996.

[LS87] LOCKEMANN, P. C. und J. W. SCHMIDT: *Datenbank-Handbuch*. Springer-Verlag, 1987.

[LS94] LIN, TEIN-HSIANG und KANG G. SHIN: *An Optimal Retry Policy Based on Fault Classification*. *Transactions on Computers*, 43(9):1014–1025, 1994.

[McK96] MCKUSICK, MARSHALL KIRK: *Secondary Storage and Filesystems*. *Computing Surveys*, 28(1):217–219, 1996.

[Mey88] MEYER, BETRAND: *Objektorientierte Software-Entwicklung*. Prentice Hall, 1988.

[MJLF84] MCKUSICK, MARSHALL K., WILLIAM N. JOY, SAMUEL J. LEFFLER und ROBERT S. FABRY: *A Fast File System for UNIX*. *Transactions on Computer Systems*, 2(3):181–197, 1984.

[MP81] MILLER, BARTON und DAVID PRESOTTO: *XOS: An Operating System for the X-Tree Architecture*. *Operating System Reviews*, 15(2):21–32, 1981.

[NW74] NEEDHAM, R. M. und M. V. WILKES: *Domains of protection and the management of processes*. *Computer Journal*, 17(2):117–120, 1974.

[NW77] NEEDHAM, R. M. und R. D. H. WALKER: *The Cambridge CAP Computer and its protection system*. Symposium on Operating System Principles, Seiten 1–10, 1977.

[OpL] *Opportunistic Locks*. [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio%2base/opportunistic\\_locks.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio%2base/opportunistic_locks.asp)  
[http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio%2base/opportunistic\\_lock\\_examples.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio%2base/opportunistic_lock_examples.asp)  
[http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio%2base/opportunistic\\_lock\\_explained.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio%2base/opportunistic_lock_explained.asp)

/base/how\_to\_request\_an\_  
opportunistic\_lock.asp http:  
//msdn.microsoft.com/library/  
default.asp?url=/library/  
en-us/fileio%base/breaking\_  
opportunistic\_locks.asp. [T+90]

[Opl65] OPLER, ASCHER: *Procedure-Oriented Language Statements to Facilitate Parallel Processing*. CACM, 8(5):306–307, 1965. [Tho96]

[Org72] ORGANICK, ELLIOT I.: *The Multics System: An Examination of Its Structure*. MIT Press, 1972. [TL93]

[Par72] PARNAS, D. L.: *On the Criteria To Be Used in Decomposing Systems into Modules*. CACM, 15(12):1053–1058, 1972. [TSF90]

[Par78] PARNAS, DAVID L.: *The Non-Problem of Nested Monitor Calls*. Operating System Reviews, 12(1):12–18, 1978. [Vah96]

PRUITT, J. L. und W. W. CASE: *Architecture of a Real Time Operating System*. Symposium on Operating System Principles, Seiten 51–59, 1975. [VGH93]

[PDZ99] PAI, VIVEK S., PETER DRUSCHEL und WILLY ZWAENEPOEL: *IO-Lite: A Unified I/O Buffering and Caching System*. OSDI, Seiten 15–28, 1999. [Vid87]

[PDZ00] PAI, VIVEK S., PETER DRUSCHEL und WILLY ZWAENEPOEL: *IO-Lite: A Unified I/O Buffering and Caching System*. Transactions on Computer Systems, 18(1):37–6, 2000. [Y+90]

[RK79] REED, DAVID P. und RAJENDRA K. KANODIA: *Synchronization with Eventcounts and Sequencers*. CACM, 22(2):115–123, 1979. [Ros94]

ROSENBLUM, MENDEL und JOHN K. OUSTERHOUT: *The Design and the Implementation of a Log-Structured File System*. Symposium on Operating System Principles, Seiten 1–15, 1991. [Ros94]

ROScoe, TIMOTHY: *Linkage in the Memesis Single Address Space Operating System*. Operating System Reviews, 28(4):48–55, 1994. [RT74]

RITCHIE, DENNIS M. und KEN THOMPSON: *The UNIX Time-Sharing System*. CACM, 17(7):365–375, 1974. [Smi88]

SMITH, JONATHAN M.: *A Survey of Process Migration Mechanisms*. Operating System Reviews, 22(3):28–40, 1988. [Str78]

STROUSTRUP, Bjarne: *On Unifying Module Interfaces*. Operating System Reviews, 12(1):90–98, 1978. [Szy98]

SZYPERSKI, CLEMENS: *Component Software*. Addison-Wesley, 1998. [TANENBAUM, ANDREW S. und OTHERS: *Experiences with the Amoeba Distributed Operating System*. CACM, 33(12):47–63, 1990. TAY, B. H. und A. L. ANANDA: *A Survey of Remote Procedure Calls*. Operating System Reviews, 24(3):68–79, 1990. THOMPSON, SIMON: *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996. THEKKATH, CHANDRAMOHAN A. und HENRY M. LEVY: *Limits to Low-Latency Communication on High-Speed Networks*. Transactions on Computer Systems, 11(2):179–203, 1993. TAM, MING-CHIT, JONATHAN M. SMITH und DAVID J. FARBER: *A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems*. Operating System Reviews, 24(3):40–67, 1990. VAHALIA, URESH: *UNIX Internals*. Prentice Hall, 1996. VOSSEN, GOTTFRIED und MARGRET GROŠHARDT: *Grundlagen der Transaktionsverarbeitung*. Addison-Wesley, 1993. VIDYASANKAR, K.: *Generalized Theory of Serializability*. Acta Informatica, 24:105–119, 1987. YOKOTE, YASUHIKO und OTHERS: *The Muse Object Architecture: A New Operating Systems Structuring Concept*. Operating System Reviews, 25(2):22–46, 1990.

Dr. Thomas Schöbel-Theuer

Verfahren zur Regelung des Datenzugriffs auf Daten  
wenigstens einer Datenspeichereinrichtung

Ansprüche

1. Verfahren zur Regulierung des Datenzugriffs bei einem aus mehreren Einzelsystemen bestehenden System auf Daten wenigstens einer Datenspeichereinrichtung, bei dem die Einzelsysteme sich freie Daten- oder Adressenbereiche der Datenquelle reservieren und die reservierten Bereiche für einen Zugriff durch andere Einzelsysteme dann gesperrt sind, wobei die Reservierungen (Locks) eine örtliche und eine zeitliche Komponente besitzen.
2. Verfahren zur Regulierung des Datenzugriffs auf Daten wenigstens einer Datenspeichereinrichtung, bei dem Adressenbereiche und/oder Teiladressenbereiche der Datenspeichereinrichtung dadurch verschoben werden, dass unter Beibehaltung der physischen Adressen die Zuordnung von logischen Adressen zu den physischen Adressen gemäß der Verschiebeoperation geändert wird.
3. Verfahren zur Regulierung des Datenzugriffs bei einem aus mehreren Einzelsystemen bestehenden System auf wenigstens

eine Datenspeichereinrichtung, bei dem die Einzelsysteme sich freie Daten- oder Adressenbereiche der Datenspeichereinrichtung reservieren und die reservierten Bereiche für einen Zugriff durch andere Einzelsysteme dann gesperrt sind, wobei gegenüber den direkt benötigten Bereichen spekulativ vergrößerte Bereiche reserviert werden.

4. Verfahren nach Anspruch 3, dadurch gekennzeichnet, dass der jeweils über den direkt benötigten Bereich hinausgehende reservierte Erweiterungsbereich bei einer entsprechenden Reservierungsanfrage seitens eines anderen Einzelsystems oder einer Datenspeichereinrichtung wenigstens teilweise freigegeben wird.

**This Page is Inserted by IFW Indexing and Scanning  
Operations and is not part of the Official Record**

**BEST AVAILABLE IMAGES**

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- BLACK BORDERS**
- IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- FADED TEXT OR DRAWING**
- BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- SKEWED/SLANTED IMAGES**
- COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- GRAY SCALE DOCUMENTS**
- LINES OR MARKS ON ORIGINAL DOCUMENT**
- REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- OTHER:** \_\_\_\_\_

**IMAGES ARE BEST AVAILABLE COPY.**

**As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.**